Deflation-based preconditioners for stochastic models of flow in porous media

by

© **Razan Abu-Labdeh**

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science.

Department of Mathematics and Statistics

Memorial University

August 2018

St. John's, Newfoundland and Labrador, Canada

Abstract

Numerical analysis is a powerful mathematical tool that focuses on finding approximate solutions to mathematical problems where analytical methods fail to produce exact solutions. Many numerical methods have been developed and enhanced through the years for this purpose, across many classes, with some methods proven to be well-suited for solving certain equations. The key in numerical analysis is, then, choosing the right method or combination of methods for the problem at hand, with the least cost and highest accuracy possible (while maintaining efficiency). In this thesis, we consider the approximate solution of a class of 2-dimensional differential equations, with random coefficients. We aim, through using a combination of Krylov methods, preconditioners, and multigrid ideas to implement an algorithm that offers low cost and fast convergence for approximating solutions to these problems. In particular, we propose to use a "training" phase in the development of a preconditioner, where the first few linear systems in a sequence of similar problems are used to drive adaptation of the preconditioning strategy for subsequent problems. Results show that our algorithms are successful in effectively decreasing the cost of solving the model problem from the cost shown using a standard AMG-preconditioned CG method.

To my loving parents and sweet husband.

Lay summary

Many real-life situations can be modelled using mathematical equations. In this thesis, I look at a mathematical problem that emerges from models related to water purification or oil extraction. Many factors affect the behaviour of these models, such as fluid pressure and the type of rocks in the subsurface. Many times, the type of rock in the subsurface is unknown, so the model pressure cannot be calculated without additional information. Thus, we usually represent these models with a kind of statistical approximation, requiring solution of many models, drawn from a statistical distribution.

There are two types of mathematical methods for finding a solution to the problem at hand, ones that give exact results and others that give approximations of the solution. For the problem presented in this thesis, exact methods take an extremely long time to compute a solution, so it is better if the second kind of methods is used. The field of mathematics that studies approximate methods is called Numerical Analysis, and the main goal of this thesis is to develop a method that can efficiently solve the model problem.

To measure which method is best, two main questions are asked:

1. How fast is an approximation to the solution produced?
2. How expensive is it to reach the best approximation?

These two questions are the main focus of study here, with the best method being the fastest to achieve acceptable accuracy.

Acknowledgements

I would like to take this opportunity to thank all those that, in one way or another, helped and supported me in writing this thesis.

I would like to begin with thanking with all my heart my parents, Dr.Abdel-Rahman and Manal, for all their love, support, encouragement, and guidance. Without them, I may have never reached where I am today. I would also like to extend my deep thanks to my husband, Abed Alsalam, for his love, patience, and support now and for years to come. I also thank my sisters (Ahlam, Ruhuf, and Hazar) and brother (Omar) for their encouragement and kindness.

I would like to thank my supervisor, Dr. Scott MacLachlan, for his suggestions, advice, contributions, and help in this research project. I would like to thank the examiners of the thesis for their time and comments. I also would like to thank Dr. Hisham Bin Zubair for all his help with the programming included in my thesis. I extend thanks for the technical support provided by Lawrence Greening of the Mathematics and Statistics department.

I wish to acknowledge the financial assistance provided by the School of Graduate Studies, Department of Mathematics and Statistics, and Natural Sciences and Engineering Research Council of Canada, in the form of graduate fellowships and teaching assistantships.

Finally, thank you to all my friends and student fellows for all their help, conversations, and good wishes through this process.

Statement of contribution

This thesis is a collaboration of work by Razan Abu-Labdeh and Dr. Scott MacLachlan. All algorithms included were developed by both parties, while the program coding used in the research and writing of thesis was done by Razan. Supervision and editing of the thesis was done by Dr.Scott MacLachlan.

Table of contents

List of tables

List of figures

Chapter 1

Introduction

The focus of this thesis is on the efficiency of numerical algorithms used to model the flow of fluids through a porous medium, a material containing pores or small holes within it. Liquid or gas can flow through such a medium, either naturally or for experimentation and research purposes. Rocks differ in composition and naturally contain pore spaces filled with gas or liquid. Many real-life applications involve flow through porous media, including management of oil reservoirs and water purification, as oil and water not only can flow through the pores in the rocks but also navigate around them. As such, models of flow through porous media are of great interest to study for their potential practical impact. In particular, finding ways of improving computer simulation algorithms for these models can contribute to real-world improvements in industrial practices. A general representation of these models consists of fluid entering a medium from one side and exiting through the opposite end. The amount of fluid and its flow through the medium is studied. We work in a two-dimensional setting in this thesis, so, a presentation of the modelled system is pictured in Figure 1.1.

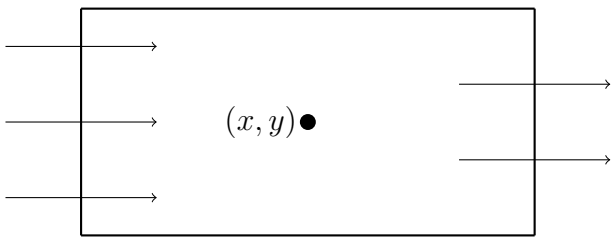


Figure 1.1: General 2D porous media model.

This system is studied in the field of Fluid Dynamics [8, 2]. Studying these systems on a reservoir scale is very complicated. In this work, we will simplify the model (considering single-phase flow), but not the complication due to unknown material properties of the porous medium, which we treat using a stochastic formulation. Such systems can be represented through one or more mathematical equations, which are then studied to find solutions. Two important equations are typically used when modelling the systems considered here, conservation of mass and Darcy's law.

To derive a general conservation law, we use L to denote the density of a conserved quantity of a fluid flowing with velocity u, and consider the time-rate of change in the amount of L in an arbitrary domain, Ω, given by

$$\frac{d}{dt}\int_\Omega L dV = -\int_{\partial\Omega}(Lu)\cdot n ds + \int_\Omega q dV,$$

where $-\int_{\delta\Omega}(Lu)\cdot n ds$ is the outward flux across the boundary of Ω and $\int_\Omega q dV$ represents external sources or sinks of L in the region. Thus, by Leibniz's rule and the divergence theorem, we get

$$\int_\Omega (\frac{\partial}{\partial t}L)dV = -\int_\Omega \nabla\cdot(Lu)dV + \int_\Omega q dV$$

$$\int_\Omega (\frac{\partial L}{\partial t} + \nabla\cdot(Lu))dV = \int_\Omega q dV.$$

So, taking L to be the mass density, ρ, and noting that Ω was an arbitrary domain, we have

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho u) = q.$$

For an incompressible fluid, where ρ is constant, this yields,

$$\nabla\cdot(\rho u) = q,$$

where $\nabla\cdot$ is the divergence operator, ρ is the density of fluid, u is the so-called Darcy velocity, and q represents external sources and sinks of fluid [8].

Darcy's Law is an emperical law resulting from Darcy's experimentation on the

flow of water in the fountains in Dijon, France in the 1850's [2]. Darcy's law is

$$u = -K\nabla p,$$

where p denotes pressure and K is called the hydraulic conductivity. By replacing the velocity in the conservation of mass equation with Darcy's law, and assuming $\rho = 1$ (or combining into K), we get

$$\nabla \cdot K\nabla p = q.$$

In some cases, it is very difficult to know the hydraulic conductivity before hand. Thus, we are left with the question of how to approximate the value of K. In this work, a statistical approach to this question will be used.

Our research problem is a two-dimensional boundary value problem with Neumann boundary conditions,

$$\begin{aligned} -\nabla \cdot (K\nabla u(x,y)) &= f(x,y), \text{ for } (x,y) \in \Omega \\ (Ku(x,y)) \cdot \overrightarrow{n} &= 0, \text{ for } (x,y) \in \partial\Omega \end{aligned}$$

where Ω is the domain and $\overrightarrow{n}$ is the outward unit normal vector on $\partial\Omega$. The problem above is explained in more detail in Chapter 3.

Analytical methods for solving differential equations can sometimes be very time consuming, and can sometimes fail to give solutions in a useful form. In such cases, numerical methods may be used to produce approximations of the true solution, where the error in approximation is acceptable, i.e. suitably small. The advantage of these methods is that they can greatly decrease the computational effort required, making them a desirable choice. Furthermore, when analytical solutions are impossible to obtain, the only approach that we have to the problem is through numerical approximation.

As a first step in numerical analysis, the differential equation is usually required to be converted from continuous to discrete form, approximating the equation via a simpler system of linear or nonlinear equations. This is achieved through applying

a discretization scheme to the continuous equation. These schemes vary in complexity and performance. The simplest of these are called Finite Difference methods, which discretize the differential equation using difference equations at each node in a mesh. More complex schemes include finite element, finite volume, and spectral discretization schemes [14, 27, 33]. Finite difference methods are presented as motivating examples in Chapter 2, while finite element methods are used for our research problem in Chapter 3.

The choice of method used to solve a problem depends on the characteristics of the problem at hand. They also vary in complexity. When choosing a suitable method to work with, two main concerns must stay in mind: is the method giving an accurate approximation, and is it doing so quickly? In measuring if our chosen method is efficient or not, we must have some recording of the effort needed to reach convergence (i.e. a good approximation) and the time spent approximating the solution. There is often a trade-off that happens with these two; that is, some methods may have high accuracy but be somewhat time-consuming. Hence, the key is to find a method with an appropriate balance.

The main cost in approximating the solution of a differential equation typically comes from solving the associated linear or nonlinear system. We discuss several numerical methods for solving such linear systems in the next chapter. Our interest is in using these methods in solving our research problem above. The main focus is on iterative methods, that start by using an initial guess of the solution and iterate (a finite number of times) until an approximation of suitable accuracy is reached. In this chapter, two common methods are presented. These are the Jacobi and Gauss-Seidel methods. Next, a sequence of numerical methods of increasing complexity and effectiveness is also given. Each algorithm is given with explanation, and the convergence and cost, both computational and storage, are discussed for each method, along with its limitations.

Chapter 3 presents more details on the specifics of the problem and algorithms developed here. In particular, the statistical technique used to generate samples of the hydraulic conductivity is given, along with details of the finite-element approach

(using the FEniCS library [49]) considered. Here, the separation of the set of systems to be solved into a "training" set and the remainder is explained, as well as how the training set is used to generate a preconditioner for the remaining systems. Examples of codes, using the PETSc and SLEPc libraries [50, 51] are included. Numerical results, discussion, and recommendations for future work given in Chapter 4.

Chapter 2

Background

In this chapter, we explore various numerical methods for solving systems of linear equations. We discuss the motive for calculating an approximate solution of the system using each method and their limitations (if any). We also present the cost of these methods.

2.1 Iterative Methods

In numerical analysis, among other aspects of math, we are often faced with solving systems of linear equations, which we will refer to as $Ax = b$ with

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & \dots & a_{1n} \\ a_{21} & a_{22} & \ddots & & a_{2n} \\ \vdots & \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{bmatrix}, \; x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix}, \text{ and } b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}. \tag{2.1}$$

These systems are often very large in size, and sometimes even so large that it is impractical to store them on a computer and solve using standard approaches. They can be extremely difficult and time consuming to solve by direct methods (such as Gaussian Elimination), thus adding unnecessary cost to the overall analysis. Hence,

iterative methods were born.

Direct methods are methods that solve linear systems in a finite number of operations, whereas, in iterative methods, the number of operations to find the exact solution may not be finite. The overall aim is to keep iterating until a reasonable approximation to the solution is reached. Another disadvantage for direct methods is that they do not efficiently deal with sparse matrices, matrices with most of their elements equal to zero. The computational cost of Gaussian Elimination for solving linear systems with dense matrices is $O(n^3)$ for an $n \times n$ matrix. While sparse direct methods offer some improvement in operations and storage cost, they are often far from competitive with modern iterative methods, which are also easier to implement on parallel computers [41].

Iterative methods are usually easy to use and, most importantly, do not consume a lot of time per iteration, making them very useful in many settings. These methods have been studied and enhanced to reach the best possible convergence rates. They have also been compared against each other in many settings, each having their strong and weak points. They can be divided into stationary and non-stationary methods. Stationary methods are iterative methods where the form of each iteration stays the same throughout the entire iteration process. Otherwise, a method is called non-stationary.

There are many different kinds of iterative methods, see [48] for a brief explanation. In this section, we will present two stationary iterative methods: Jacobi and Gauss-Seidel, and give a variation of Gauss-Seidel. A discussion of convergence will also be presented briefly at the end of the section.

Jacobi methods

The Jacobi method was introduced by Carl Jacob Jacobi. It is considered one of the easiest iterative methods for solving a system of linear equations in the form of $Ax = b$.

To apply the Jacobi method to the system in (2.1), we rewrite the first equation to solve for x_1, the second equation for x_2, the third for x_3, and so forth, to get the following set of equations:

$$\begin{aligned} x_1 &= \tfrac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n) \\ x_2 &= \tfrac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n) \\ &\vdots \\ x_n &= \tfrac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n(n-1)}x_{(n-1)}). \end{aligned} \tag{2.2}$$

Let the vector $x^{(0)} = [x_1^{(0)}, x_2^{(0)} \dots x_n^{(0)}]^T$ be an initial guess for the solution. Using $x^{(0)}$ as an input into the right-hand side of the set of equations in (2.2) , we compute $x^{(1)} = [x_1^{(1)}, x_2^{(1)} \dots x_n^{(1)}]^T$, where the components of $x^{(1)}$ are given by the left-hand side of (2). By this, we will have completed the first iteration of the method, getting the first approximation, $x^{(1)}$. We repeat the process again, using the first approximation as inputs in the right-hand side of the equations to obtain the next approximation, $x^{(2)}$. Repetition is then continued, until we eventually reach some measure of convergence. In general, the iteration takes the form:

$$\begin{aligned} x_1^{(k+1)} &= \tfrac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1n}x_n^{(k)}) \\ x_2^{(k+1)} &= \tfrac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2n}x_n^{(k)}) \\ &\vdots \\ x_n^{(k+1)} &= \tfrac{1}{a_{nn}}(b_n - a_{n1}x_1^{(k)} - a_{n2}x_2^{(k)} - \cdots - a_{n(n-1)}x_{(n-1)}^{(k)}), \end{aligned} \tag{2.3}$$

where the superscript k denotes the previous iteration and $k+1$ is the current iteration, giving vectors $x^{(k)}$ as the previous approximation and $x^{(k+1)}$ as the next approximation. Convergence, simply put, is reaching an approximate solution that is close enough to the exact solution. It should guarantee that the error in our approximation is small in comparison to the discretization error of the underlying differential equation, given by the difference between the exact and approximate solutions, $e = exact - approximation$.

This method requires the entries on the diagonal of A to be all non-zero elements. To simplify the explanation, we will present the matrix form of the method. Writing

$A = D - L - U$, where D is the diagonal of matrix A and L, U are the strictly lower and upper triangular parts of A, respectively, we can then write out the iteration as in [7] by

$$x^{(k+1)} = R_J x^{(k)} + D^{-1} b, \tag{2.4}$$

where $R_J = D^{-1}(L + U)$.

Gauss-Seidel

The Gauss-Seidel (GS) method was first introduced by two German mathematicians, Carl Friedrich Gauss and Philipp Ludwig von Seidel. It is an iterative method similar to Jacobi's method, but with important differences.

Take the same linear system (2.1) from above with the same requirement that the diagonal elements of A be nonzero. In Jacobi, it is clear that any entry in $x^{(k+1)}$ can be obtained once all the components of $x^{(k)}$ are calculated but without any other components of $x^{(k+1)}$. Gauss-Seidel is an alternative to that process, updating the components of $x^{(k+1)}$ using components of $x^{(k)}$ that have been previously calculated, as well as those of $x^{(k+1)}$ that are already known. This can be seen from the component form of the method in the following:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \tag{2.5}$$

for $i = 1, 2, \ldots, n$. The reuse of entries in $x^{(k+1)}$ is clear here from the first summation, where we use $x_j^{(k+1)}$, for $j = 1, 2, \ldots i - 1$ to find $x_i^{(k+1)}$. This is the main difference between Jacobi and Gauss-Seidel. For clarity, writing out the n equations from (2.5) gives:

$$\begin{aligned}
x_1^{(k+1)} &= \tfrac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1n}xn^{(k)}) \\
x_2^{(k+1)} &= \tfrac{1}{a_{22}}(b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - \cdots - a_{2n}x_n^{(k)}) \\
\vdots &= \vdots \\
x_n^{(k+1)} &= \tfrac{1}{a_{nn}}(b_n - a_{n1}x_1^{(k+1)} - a_{n2}x_2^{(k+1)} - \cdots - a_{n(n-1)}x_{(n-1)}^{(k+1)})
\end{aligned}$$

Again, we can write Gauss-Seidel in matrix form [7],

$$x^{(k+1)} = (D - L)^{-1}[Ux^{(k)} + b], \tag{2.6}$$

where L, D and U are defined as above.

A further method based on Gauss-Seidel is the weighted Gauss-Seidel method, often called successive over relaxation (SOR). Simply speaking, it incorporates a relaxation parameter, ω, into the GS method. Eigenvalue analysis is usually needed to find the best parameter ω, which gives the best improvement to the rate of convergence of the iteration process. In component form, the SOR method is:

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}}\Big(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\Big) + (1-\omega)x_i^{(k)}, \tag{2.7}$$

or, as a system,

$$\begin{aligned}
x_1^{(k+1)} &= \tfrac{\omega}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1n}x_n^{(k)}) + (1-\omega)x_1^{(k)} \\
x_2^{(k+1)} &= \tfrac{\omega}{a_{22}}(b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - \cdots - a_{2n}x_n^{(k)}) + (1-\omega)x_2^{(k)} \\
\vdots &= \vdots \\
x_n^{(k+1)} &= \tfrac{\omega}{a_{nn}}(b_n - a_{n1}x_1^{(k+1)} - a_{n2}x_2^{(k+1)} - \cdots - a_{n(n-1)}x_{(n-1)}^{(k+1)}) + (1-\omega)x_n^{(k)}
\end{aligned}$$

A detailed proof of how to derive SOR from GS can be found in [42]. It is quite clear that if $\omega = 1$, then the last term of the equations above vanishes and we end up with GS. It can be viewed in matrix form as:

$$(D - \omega L)x^{(k+1)} = [(1-\omega)D + \omega U]x^{(k)} + \omega b. \tag{2.8}$$

In most literature, the case $\omega < 1$ is referred to as under-relaxation, and the case $\omega > 1$ is over-relaxation. ω can also, in some cases, be taken to be different at every

iteration, so the iterative method would use a set of $\{\omega_1, \omega_2, \dots\}$ [48].

Kahan's theorem is an interesting result, presenting the essential choice of ω. It states that if SOR converges then $0 < \omega < 2$. Choosing ω outside this interval will result in divergence of SOR. The analysis of SOR and the choice of ω is discussed in detail in [48].

To compare these methods to one another, we introduce some common notation. Any stationary one-step iterative method to solve $Ax = b$ can be written as

$$Mx^{(k+1)} = Nx^{(k)} + b,$$

where b is the right-hand side vector, and M is a non-singular matrix with $A = M - N$. Writing

$$x^{(k+1)} = (M^{-1}N)x^{(k)} + \hat{b}, \tag{2.9}$$

where $\hat{b} = M^{-1}b$, gives the iteration matrix $M^{-1}N$. In Jacobi and GS, $M_J^{-1}N_J = D^{-1}(L+U)$ and $M_G^{-1}N_G = (D-L)^{-1}(U)$. Note that what we refer to as $R_{(J)}$ in (2.4) is the same as $M_J^{-1}N_J$.

For comparison between these methods, tools such as the spectral radius and rate of convergence are needed. The spectral radius is determined by the eigenvalues, λ , of a matrix; for any square matrix, G,

$$\rho(G) := \max\{|\lambda|\colon \lambda \in \lambda(G)\}$$

This is particularly important when studying the convergence of iterative methods. The error of the k^{th} iteration can be expressed as $e^{(k)} = x - x^{(k)}$, where x is the exact solution (or true solution) and $x^{(k)}$ is the approximated solution after k iterations. It is natural that if, after k iterations we have reached x, then $x^{(k)} = x$ and, normally, the next iteration $x^{(k+1)} = x$ so Equation (2.9) can be written as

$$x = (M^{-1}N)x + \hat{b}, \tag{2.10}$$

which can also be derived directly from $Ax = (M - N)x = b$. Subtracting Equation (2.10) from (2.9) yields

$$\begin{aligned} x^{(k+1)} - x &= (M^{-1}N)[x^{(k)} - x] \\ e^{(k+1)} &= (M^{-1}N)e^{(k)}. \end{aligned}$$

Repeating this process k times, each time decreasing the index in the superscript by 1, we get

$$e^{(k+1)} = (M^{-1}N)e^{(k)} = (M^{-1}N)^2e^{(k-1)} = \cdots = (M^{-1}N)^{k+1}e^{(0)}.$$

Note that, in most practical problems, we have no idea what $e^{(0)}$ is (otherwise $\mathbf{x}$ would be known, and there would not be a need to iterate any more).

Using simple matrix norms, however, we get

$$||e^{(k)}|| \leq ||(M^{-1}N)^k||.||e^{(0)}||.$$

This shows that if $||(M^{-1}N)^k|| \to 0$ then $||e^{(k)}|| \to 0$, and the method converges.

The closer $||(M^{-1}N)^k||$ is to zero, the faster $||e^{(k)}|| \to 0$, i.e, the faster convergence is guaranteed to be. Alternately, the closer it is to one, the slower the convergence. Note that $||(M^{-1}N)^k||$ may be greater than or equal to 1 as well, indicating that the method is diverging, and that the error can get larger as iterations proceed.

When $M^{-1}N$ is diagonalizable, $||M^{-1}N|| = |\lambda_{max}| = \rho(M^{-1}N)$, and $||(M^{-1}N)^k|| = \rho(M^{-1}N)^k$ so, instead of taking these matrix norms, it is easier to deal with eigenvalues. If $\rho(M^{-1}N) < 1$, the method converges.

An $n \times n$ matrix, A, is diagonally dominant when $\sum_{\substack{j=1 \\ i \neq j}}^{n} \frac{|a_{ij}|}{a_{ii}} \leq 1$. Strictly diagonal dominant just means the inequality is strictly less than instead of allowing equality. By direct calculation, the ij^{th} entry in the Jacobi iteration matrix, $D^{-1}(L + U)$, is zero if $i = j$ and $-a_{ij}/a_{ii}$ otherwise. Thus, Gerschgorin's theorem guarantees that the

Jacobi method converges, $\rho(D^{-1}(L+U)) < 1$, if A is strictly diagonally dominant. Usually, the more dominant the diagonal is (the smaller $\sum_{j\neq i}^{n} \frac{|a_{ij}|}{a_{ii}}$ is), the more rapid the convergence [16].

A square matrix, A, is symmetric if $a_{ij} = a_{ji}, j \neq i$ holds and it is positive-definite if $x^T Ax > 0$ for any $n \times 1$ real-valued vector, $x \neq 0$, when A is real-valued. The GS method is guaranteed to converge, $\rho((D-L)^{-1}(U)) < 1$, if A is an $n \times n$ symmetric and positive-definite matrix. This can be found in more detail in [47, Section 3.4] through the discussion of the Ostrowski-Reich theorem.

Here is where SOR can be an attractive method in comparison to GS. In many cases, the spectral radius, $\rho((D-L)^{-1}(U))$, is close to 1. This results in the GS method being slow to converge. By choosing the SOR weight, $\omega \neq 1$, we can make $\rho((D-\omega L)^{-1}((1-\omega)D+\omega U))$ potentially much smaller than $\rho((D-L)^{-1}U)$. Another tool for comparisons is the rate of convergence. In simple terms, it is the speed at which $||(M^{-1}N)^k||$ will go to zero.

The average rate of convergence, denoted by $R(.)$ can be defined as in [47] for an $n \times n$ matrix, A, with $||A^k|| < 1$ as

$$R(A^k) := -\frac{1}{k} \ln ||A^k||,$$

and the asymptotic rate of convergence, denoted by $R_\infty(.)$, is

$$R_\infty(A^k) := \left| \lim_{k\to\infty} R(A^k) \right| = -\ln \rho(A).$$

Following [47], if $R(A^k) < R(B^k)$ for matrices A and B, then B is iteratively faster than A.

The average rate of reduction in error per iteration after k iterations is denoted by

$$\sigma := \left(\frac{||e^{(k)}||}{||e^{(0)}||} \right)^{\frac{1}{k}}.$$

So, since $||e^{(k)}|| \leq ||e^{(0)}||||(M^{-1}N)^k||$ then

$$\sigma \leq ||(M^{-1}N)^k||^{\frac{1}{k}} = e^{-R\left((M^{-1}N)^k\right)},$$

which means that the average rate of reduction after k iterations is bounded above by exponential decay at the average rate of convergence after k iterations [47].

Now, with these definitions, we can present some comparison between GS and Jacobi.

Theorem (Stein-Rosenberg) [31]: If each $a_{ij} \leq 0$ for $i \neq j$, and A has non-negative diagonal elements, then one and only one of following statements holds:

1. $\rho(M_G^{-1}N_G) = \rho(M_J^{-1}N_J) = 0.$
2. $\rho(M_G^{-1}N_G) = \rho(M_J^{-1}N_J) = 1.$
3. $0 < \rho(M_G^{-1}N_G) < \rho(M_J^{-1}N_J) < 1.$
4. $1 < \rho(M_J^{-1}N_J) < \rho(M_G^{-1}N_G).$

This theorem presents the relationship between the spectral radii of both Jacobi and GS. The interesting thing they show is that, under these assumptions, Jacobi and GS either both converge or both diverge. Point 3 is of particular importance; saying that $\rho(M_G^{-1}N_G)$ is closer to zero than $\rho(M_J^{-1}N_J)$, stating that GS is faster to converge than Jacobi when both converge. In other words, $R_\infty(M_G^{-1}N_G) > R_\infty(M_J^{-1}N_J)$. In fact, in many relevant cases, what takes Jacobi 2 iterations to accomplish, takes GS only one iteration to do [47].

We note that the condition on A in the theorem is important. If the diagonal is non-positive, then the theorem does not apply any more and, in fact, we can find counterexamples where one method converges while the other diverges.

As for the average rate of convergence, a simple example seen in [47] shows that, for a single iteration, the rate of convergence for Jacobi can be faster than that for GS.

So, we are unable to generalize that $R(M_G^{-1}N_G) > R(M_J^{-1}N_J)$ for all iterations.

Storage: From the explanation of both Jacobi and GS methods above, one naturally expects to need more storage space for Jacobi due to the fact that, in GS, the method overwrites its components. In [7], Jacobi is shown to need storage of $2n$ real values for the approximation vectors, $x^{(k)}$ while, in GS, it is reduced to only n real values.

While both Jacobi and GS are powerful stationary iterative methods, they both have their strong and weak points. We cannot say that GS is always better to use than Jacobi. In fact, for some systems, GS is the wrong choice of method and is found to be completely useless, even diverging in some cases. They are both tools used to this day in numerical analysis among other stationary and non-stationary methods.

We do note that while the GS method is often faster to converge than the Jacobi method, both methods are still generally very slow to converge. This is the reason why these methods are rarely used alone in solution but are excellent choices as relaxation methods in multigrid methods, which will be discussed in detail later on in the chapter.

2.2 Lanczos, Conjugate Gradient and Deflation

In the previous section, stationary iterative methods were shown to use previous approximate solutions, $x^{(k)}$, to find the next approximate solution, $x^{(k+1)}$. We start this section by introducing non-stationary polynomial methods.

Recall, the residual of system (2.1) is $r = b - Ax$. If the initial guess is used, $x^{(0)}$, we write $r^{(0)} = b - Ax^{(0)}$. Substituting b with Ax, we get

$$r^{(0)} = A(x - x^{(0)}).$$

Recall the error is defined as $e^{(i)} = x - x^{(i)}$. The residual above then becomes

$$r^{(0)} = Ae^{(0)}.$$

Generally speaking,

$$r^{(i)} = Ae^{(i)}. \tag{2.11}$$

This is an important equation, showing the relationship between the residual of a system and the error at the i^{th} iteration.

Another family of iterative methods can be written as

$$x^{(k+1)} = x^{(k)} + \omega_{k+1} r^{(k)},$$

where $r^{(k)}$ and ω_{k+1} are the residual at the k^{th} step, and the weight that represents the step size at step $(k+1)$, respectively. So,

$$\begin{aligned}
x^{(1)} &= x^{(0)} + \omega_1 r^{(0)} \\
&= x^{(0)} + \omega_1 (b - Ax^{(0)}) \\
x^{(2)} &= x^{(1)} + \omega_2 r^{(1)} \\
&= x^{(1)} + \omega_2 (b - Ax^{(1)}) \\
\vdots &= \vdots \\
x^{(k+1)} &= x^{(k)} + \omega_{k+1} r^{(k)} \\
&= x^{(k)} + \omega_{k+1} (b - Ax^{(k)}).
\end{aligned}$$

If we subtract both sides from x ,the true solution, we obtain

$$x - x^{(k+1)} = x - [x^{(k)} + \omega_{k+1}(b - Ax^{(k)})] e^{(k+1)} = e^{(k)} - \omega_{k+1}(b - Ax^{(k)}).$$

Substituting (2.11) in the last step, we reach

$$e^{(k+1)} = e^{(k)} - \omega_{k+1} Ae^{(k)} = (I - \omega_{k+1} A) e^{(k)}. \tag{2.12}$$

Retracing our steps by substituting $e^{(k)} = (I - \omega_k A) e^{(k-1)}$ in (2.12) we get

$$e^{(k+1)} = (I - \omega_{k+1} A)(I - \omega_k A) e^{(k-1)}.$$

Repeating this $(k-2)$ times, we have

$$e^{(k+1)} = \prod_{i=1}^{k+1}(I - \omega_i A)e^{(0)}. \tag{2.13}$$

Defining $P_{k+1}(z) = \prod_{i=1}^{k+1}(I - \omega_i z)$, we write $P_{k+1}(A) = \prod_{i=1}^{k+1}(I - \omega_i A)$ as a polynomial in matrix A. Hence, a method that can be written as $e^{(k+1)} = P_{k+1}(A)e^{(0)}$ is called a polynomial method.

An interesting observation is in the following theorem.

Theorem 1 *$x^{(k)} - x^{(0)}$ can be written as a combination of $A^i r^{(0)}$, for $i = 0, 1, \ldots, (k-1)$.*

Proof:
If we work a bit with $x^{(i+1)} = x^{(i)} + \omega_{i+1} r^{(i)}$ iteratively, we have

$$\begin{aligned}
x^{(1)} &= x^{(0)} + \omega_1 r^{(0)} \\
x^{(2)} &= x^{(0)} + \omega_1 r^{(0)} + \omega_2 r^{(1)} \\
&= x^{(0)} + \omega_1 r^{(0)} + \omega_2 (b - Ax^{(1)}) \\
&= x^{(0)} + \omega_1 r^{(0)} + \omega_2 (b - A(x^{(0)} + \omega_1 r^{(0)})) \\
&= x^{(0)} + \omega_1 r^{(0)} + \omega_2 (b - Ax^{(0)} - \omega_1 A r^{(0)}) \\
&= x^{(0)} + \omega_1 r^{(0)} + \omega_2 (r^{(0)} - \omega_1 A r^{(0)}) \\
&= x^{(0)} + (\omega_1 + \omega_2) r^{(0)} - \omega_1 \omega_2 A r^{(0)}.
\end{aligned} \tag{2.14}$$

For $x^{(3)}$, by the same procedure, we have

$$x^{(3)} = x^{(0)} + (\omega_1 + \omega_2 + \omega_3) r^{(0)} + (-\omega_1\omega_2 - \omega_1\omega_3 - \omega_2\omega_3) A r^{(0)} + \omega_1\omega_2\omega_3 A^2 r^{(0)}.$$

Note, since $\{\omega_i\}_{i=1}^{k+1}$ are just weights, one can consider $(\omega_1 + \omega_2 + \omega_3)$ as one constant, say $\tilde{c}_1$,$(-\omega_1\omega_2 - \omega_1\omega_3 - \omega_2\omega_3)$ as $\tilde{c}_2$, and $\omega_1\omega_2\omega_3$ as $\tilde{c}_3$. So, $x^{(3)}$ becomes $x^{(3)} = x^{(0)} + \tilde{c}_1 r^{(0)} + \tilde{c}_2 A r^{(0)} + \tilde{c}_3 A^2 r^{(0)}$.

By induction, if

$$x^{(k)} = x^{(0)} + \tilde{c}_1 r^{(0)} + \tilde{c}_2 A r^{(0)} + \tilde{c}_3 A^2 r^{(0)} + \cdots + \tilde{c}_{k-1} A^{k-1} r^{(0)}.$$

Then

$$Ax^{(k)} = A(x^{(0)} + \tilde{c}_1 r^{(0)} + \tilde{c}_2 A r^{(0)} + \tilde{c}_3 A^2 r^{(0)} + \cdots + \tilde{c}_{k-1} A^{k-1} r^{(0)}).$$

So, by using $x^{(k+1)} = x^{(k)} + \omega_{k+1} r^{(k)}$, we get

$$x^{(k+1)} = x^{(0)} + c_1 r^{(0)} + c_2 A r^{(0)} + c_3 A^2 r^{(0)} + \cdots + c_k A^k r^{(0)},$$

where

$$c_j = \begin{cases} \tilde{c}_1 + \omega_{k+1}, & \text{for } j = 1 \\ \tilde{c}_j - \omega_{k+1}\tilde{c}_{j-1}, & \text{for } 1 < j < k \\ -\omega_{k+1}\tilde{c}_{k-1}, & \text{for } j = k. \end{cases}$$

□

This is the idea of Krylov space methods. A Krylov space is a space spanned from the set of vectors : $\{A^0 r^{(0)}, A^1 r^{(0)}, A^2 r^{(0)}, \ldots, A^{k-1} r^{(0)}\}$. The number of vectors in the set is k, thus, the space resulting from spanning this set is typically of dimension k. It is denoted as

$$K_k(A, r^{(0)}) = span\{r^{(0)}, Ar^{(0)}, A^2 r^{(0)}, A^3 r^{(0)}, \ldots, A^{k-1} r^{(0)}\}. \tag{2.15}$$

With each iteration taken the dimension of the respective Krylov space increases by one.

There exist a large number of methods that aim at finding the best approximation of the form $x^{(k)} = x^{(0)} + y_k$, for $y_k \in K_k(A, r^{(0)})$. These methods are classified as Krylov methods. In some resources, they are also referred to as Krylov projections.

These methods can be classified into 4 main categories [46]:

1-Ritz-Galerkin.
2-Minimum residual norm approach.
3-Petrov-Galerkin.
4-Minimum error norm approach.

The Ritz-Galerkin approach leads to both the Lanczos and conjugate gradient methods, while the minimum residual norm approach gives the MINRES and GMRES methods [46].

Arnoldi Method:

In 1951, the Arnoldi method was first introduced. It aims to compute an orthonormal basis set that spans the space $K_k(A, r^{(0)})$. Recall, the vectors in $\{v_1, v_2, \ldots, v_s\}$ are orthonormal if $v_i \perp v_j$, $\forall i \neq j$, and $||v_i||= 1$, $\forall i = 1, 2, \ldots, s$. Before we present Arnoldi's algorithm, we define Hessenberg matrices.
Definition: An upper Hessenberg matrix $(H_k)_{ij}$ is a square matrix with entries

$$(H_k)_{ij} = \begin{cases} h_{ij},\ j = 1, 2 \ldots, k,\ i = 1, 2, \ldots, \min(j+1, k) \\ 0,\ \text{otherwise} \end{cases}$$

In other words, an upper Hessenberg matrix has all entries under the first sub diagonal equal to zero. For example, the following is an upper Hessenberg matrix:

$$\begin{bmatrix} X & X & X & X \\ X & X & X & X \\ 0 & X & X & X \\ 0 & 0 & X & X \end{bmatrix}.$$

Arnoldi's algorithm is given in the following.

Arnoldi's algorithm:

Let q_1 be a given normalized vector ($||q_1||= 1$):

For $j = 1, 2, 3, \ldots, (k-1)$
$\{\hat{q}_{j+1} := Aq_j$
For $i = 1, 2, 3, \ldots, j$
$\{h_{ij} = q_i^T \hat{q}_{j+1}$
$\hat{q}_{j+1} = \hat{q}_{j+1} - h_{ij} q_i$
$\}$
$h_{j+1,j} = ||\hat{q}_{j+1}||$
$q_{j+1} = \frac{\hat{q}_{j+1}}{h_{j+1,j}}$
$\}$

The algorithm outputs a set of vectors $\{q_1, q_2, \ldots\}$. These form a basis of the Krylov space, $K_k(A)$. They are clearly all normalized vectors, due to normalization in the final step in the algorithm. They are also all orthogonal to one another, because of the orthogonalization step, $\hat{q}_{j+1} = \hat{q}_{j+1} - h_{ij} q_i$. Hence, the vectors in the set $\{q_1, q_2, \ldots\}$ are orthonormal. They also span $K_k(A)$. By induction, it can be shown that q_j can be written as $p_{j-1}(A)q_1$, where p_{j-1} is a polynomial of degree $(j-1)$ [39].

We note the obvious break-down point of the algorithm is when $h_{j+1,j} = 0$. Hence, $\hat{q}_{j+1} = 0$ at this point. From the algorithm, this says that $\hat{q}_{j+1} = Aq_j - \sum_{i=1}^{j} h_{ij} q_i = 0$, implying that q_j is the last possible independent vector that can be added to the set $\{q_1, q_2, \ldots, q_j\}$ that spans the subspace, making the subspace $K_j(A)$ invariant under A [39]. Since $Aq_j = \sum_{i=1}^{j} h_{ij} q_i$, $Aq_j \in$ span $\{ q_1, q_2, \ldots, q_j\}$.

Theorem 2 *Given* $q_1 = \frac{1}{||r^{(0)}||} r^0$,
then span$\{q_1, q_2, \ldots, q_{j+1}\}$ = *span*$\{r^{(0)}, Ar^{(0)}, A^2 r^{(0)}, \ldots, A^j r^{(0)}\}$.

Proof:
We prove this by induction. Let $q_1 = \frac{r^{(0)}}{||r^{(0)}||}$.

(1) For $j = 1$, by our assumption, it is clear that $r^{(0)} = ||r^{(0)}||q_1$ and span$\{q_1\}$ = span$\{r^{(0)}\}$.

(2) Now we prove the result for the general case, j. Assume the result holds for

$i = 1, 2 \ldots j$,

$$\text{span}\{q_1, q_2, \ldots, q_i\} = \text{span}\{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \ldots, A^{i-1}r^{(0)}\}.$$

In particular, this means that there are constants $\alpha_1, \ldots, \alpha_j$ and $\beta_1, \ldots, \beta_j$ such that $q_j = \alpha_1 r^{(0)} + \alpha_2 Ar^{(0)} + \cdots + \alpha_j A^{j-1}r^{(0)}$ and $A^{j-1}r^{(0)} = \beta_1 q_1 + \beta_2 q_2 + \cdots + \beta_j q_j$.

For the first direction, we multiply A into $q_j = \alpha_1 r^{(0)} + \alpha_2 Ar^{(0)} + \cdots + \alpha_j A^{j-1}r^{(0)}$ to get $Aq_j = \alpha_1 Ar^{(0)} + \alpha_2 A^2 r^{(0)} + \cdots + \alpha_j A^j r^{(0)}$. By the orthogonalization step in Arnoldi, $\hat{q}_{j+1} = Aq_j - \sum_{i=1}^{j} h_{ij} q_i$. From above, we can write Aq_j as a linear combination of the vectors $A^i r^{(0)}$. By induction, we can do the same for each q_i. So, $q_{j+1} = \alpha_1 r^{(0)} + \cdots + \alpha_{j+1} A^j r^{(0)}$ for some constants, $\alpha_1, \ldots$, α_{j+1}.

For the reverse inclusion, the same process can be used to show $A^j r^{(0)} = \beta_1 q_1 + \beta_2 q_2 + \cdots + \beta_{j+1} q_{j+1}$ for some constants β_1, $\ldots$, β_{j+1} .

□

We can now say that $K_j(A, r^{(0)})$=span$\{q_1, q_2, q_3, \ldots, q_j\}$. We denote the matrix Q_k with columns consisting of the vectors in the set $\{q_1, q_2, \ldots, q_k\}$ from the Arnoldi algorithm, and $H_{k,(k-1)}$ is the upper Hessenberg matrix with elements h_{ij}. Now, the Arnoldi algorithm [46] gives the relation

$$AQ_{k-1} = Q_k H_{k,(k-1)}.$$

This is the matrix form of $Aq_j = \sum_{i=1}^{j+1} h_{ij} q_i$, for $j = 1, 2, \ldots, k$. A simpler way of writing this relation is

$$AQ_k = Q_k H_k + S, \tag{2.16}$$

for $S = (q_{k+1} \cdot h_{(k+1),k}) e_k^T$, where e_k is the unit vector of length k with a 1 in its last position. We note that S is $n \times k$, A is $n \times n$, Q_k is $n \times k$ and H_k is a $k \times k$ matrix. Since the basis vectors are orthogonal, $Q_k^T q_{k+1} = 0$. Thus, (2.16) implies that [36]

$$Q_k^T A Q_k = H_k. \tag{2.17}$$

Cost: For computational cost, Arnoldi does one vector update of $\hat{q}_{j+1}$ for $j = 1$, two updates for $j = 2$, the next iteration, three for the next iteration and so forth. For each iteration, the method computes 1 matrix-vector multiplication Aq_j. The major cost of the algorithm comes from this multiplication. As for accumulated computation, it performs 1 matrix vector multiplication and then orthogonalization adding up to overall computational cost of $2nm^2 + m * (\text{ No of nonzeros}(A))$ for m iterations. Overall, Arnoldi is somewhat cheap in comparison with other methods with a cost of $O(m^2 n)$. As for storage, the only additional vector to be stored at iteration j is q_{j+1}. For accumulated storage over m iterations, the method stores the Hessenberg matrix and the matrix Q consisting of the vectors $\{q_1, q_2, \ldots, q_m\}$, giving total cost of $nm + \frac{1}{2}m^2$ doubles, where nm is the storage for Q and $\frac{1}{2}m^2$ is the estimated storage for $H_k = \sum_{j=1}^{k}(j+1)$ [39].

Lanczos Method:

A more efficient method was developed after the Arnoldi method. It is, in essence, an extension of Arnoldi, with the additional condition that the matrix A be a symmetric matrix. Lanczos developed this method in the 1950's. When first constructed, the algorithm was intended to find the eigenvalues and eigenvectors of a symmetric matrix. It was later used as an efficient way to find an orthonormal basis of the Krylov space. In 1950, Lanczos also published a paper on the non-symmetric Lanczos method for non-symmetric matrices [36].

Theorem 3 *If A is symmetric matrix, then the upper Hessenberg matrix $H_k = Q_k^T A Q_k$ is a symmetric and tridiagonal matrix.*

Proof:
The proof is fairly straight forward. Since A is symmetric, $A^T = A$. So, using simple matrix properties, we get

$$\begin{aligned} H_k &= Q_k^T A Q_k, \\ (H_k)^T &= (Q_k^T A Q_k)^T \\ &= (Q_k)^T (A)^T (Q_k^T)^T \\ &= Q_k^T A Q_k. \end{aligned}$$

Hence, $H_k = H_k^T$ meaning H_k is symmetric. Because of the upper triangular structure of the Hessenberg matrix, H_k, if it is a symmetric matrix then it is also tridiagonal.

□

The Lanczos algorithm is given in the following.

Lanczos Algorithm: [38]

Let q_1 be any given normalized vector with $||q_1||= 1$. Let $\beta_1 = 0$, $q_0 = 0$
For $j = 1, 2, \ldots$
$\{w_j = Aq_j - \beta_j q_{j-1}$
$\alpha_j = w_j^T q_j$
$w_j = w_j - \alpha_j q_j$
$\beta_{j+1} = ||w_j||$
$q_{j+1} = \frac{w_j}{\beta_{j+1}}$.
$\}$

The advantage of this method over Arnoldi is that it only needs to save 3 vectors, w_j, q_j and q_{j+1}. As for computation, it requires only 1 dot product, for α_j, and 1 AXPY (matrix-vector multiply plus vector), w_j, giving an overall cost of O(n) per iteration, independent of how many iterations have been performed.

Applying Lanczos with $j = 1$, we can see that

$$\begin{aligned} w_1 &= Aq_1 \\ \alpha_1 &= w_1^T q_1 \\ &= q_1^T A q_1 \\ w_1 &= Aq_1 - w_1 q_1 \\ \beta_2 &= ||w_1|| \\ q_2 &= \frac{w_1}{||w_1||}. \end{aligned} \tag{2.18}$$

If we were to compare with the elements of H_k from $Q_k^T A Q_k = H_k$, we can find

some relations between them. First, from the second line in the algorithm, we see that $q_1^T A q_1$ is equivalent to h_{11}. Secondly, from the tridiagonal structure of H_k, we can conclude that $h_{13} = h_{14} = \cdots = 0$. Thirdly, from symmetry of H_k, we get $h_{j-1,j} = h_{j,(j-1)}$. By direct computation, $\beta_2 = ||Aq_1 - (q_1^T A q_1)q_1||$ is equal to $h_{12} = h_{21}$. So, in other words, for the first step of the algorithm, we have:

1- the diagonal of H_k, h_{jj}, equals α_j.
2- $h_{ij} = 0$ for $i = 1, 2, \ldots, (j-2)$.
3- $h_{j-1,j} = h_{j,(j-1)} = \beta_j$.

We denote the symmetric form of the $(k+1) \times k$ matrix $H_{k+1,k}$ by $\hat{T}_k$,

$$\hat{T}_k = \begin{bmatrix} \alpha_1 & \beta_2 & & & & & \\ \beta_2 & \alpha_2 & \beta_3 & & & & \\ & \beta_3 & \alpha_3 & \beta_4 & & & \\ & & \ddots & \ddots & & & \\ & & & & & & \\ & & & \beta_{k-1} & \alpha_{k-1} & \beta_k & \\ & & & & \beta_k & \alpha_k & \\ & & & & & \beta_{k+1} & \end{bmatrix}. \tag{2.19}$$

From Lanczos, we have $AQ_k = Q_{k+1}\hat{T}_k$. In polynomial methods, we also saw the solution can be written as

$$x^{(k)} = x^{(0)} + P_k(A)r^{(0)}. \tag{2.20}$$

If we write $V_k = [r^{(0)}, Ar^{(0)}, \ldots, A^{k-1}r^{(0)}]$, we get $x^{(k)} = x^{(0)} + V_k y_k$, where $y_k \in \mathbb{R}^k$, and, after Arnoldi, we are able to use $Q_k = [q_1, q_2, ..., q_k]$ to write (2.20) as

$$x^{(k)} = x^{(0)} + Q_k y_k. \tag{*}$$

From here, one approach is to find the $x^{(k)}$ that will result in the smallest residual in norm, such that $x^{(k)} - x^{(0)} \in K_k(A, r^{(0)})$. Using the above,

$$\min \ ||b - Ax^{(k)}|| \ = \min \ ||b - A(x^{(0)} + Q_k y_k)||$$

$$
\begin{aligned}
&= \min \ ||b - Ax^{(0)} - AQ_k y_k|| \\
&= \min \ ||r^{(0)} - AQ_k y_k|| \\
&= \min \ ||r^{(0)} - Q_{k+1}\hat{T}_k y_k|| \\
&= \min \ ||Q_{k+1}^T r^{(0)} - \hat{T}_k y_k||.
\end{aligned}
$$

This leads to a method called MINRES. We omit the details of this method here since we will not use it. Taking a Ritz-Galerkin approach, requiring $Q_k^T r^{(k)} = 0$, leads to another method that we can use to solve sparse symmetric linear systems, called the conjugate gradient algorithm (CG).

Conjugant gradient:

The set of vectors, $\{r^{(i)}\}_{i=1}^{k}$, that form the residual at each iteration in the CG method are forced to be orthogonal to one another, i.e $(r^{(i)}, r^{(j)}) = 0$ for all $i \neq j$ [21]. So, if we were to enforce that $r^{(k)}$ be orthogonal to every vector in the range of Q_k, then

$$
\begin{aligned}
Q_k^T r^{(k)} &= 0 \\
Q_k^T (b - Ax^{(k)}) &= 0 \\
Q_k^T (b - A(x^{(0)} + Q_k y_k)) &= 0 \\
Q_k^T (r^{(0)} - AQ_k y_k) &= 0 \\
Q_k^T r^{(0)} &= Q_k^T AQ_k y_k.
\end{aligned}
$$

Defining $T_k = Q_k^T AQ_k$ to be a tridiagonal matrix that matches $\hat{T}_k$ but without the last row, making T_k a $(k \times k)$ matrix, then this can be written as $T_k y_k = Q_k^T r^{(0)}$. Note that T_k is also symmetric and positive definite if A is.

We now derive the conjugate gradient method from Lanczos in the case that A is positive definite. We follow the method presented in [30]. Start by performing a Cholesky decomposition of T_k. Let $T_k = L_k U_k$, where

$$L_k = \begin{bmatrix} 1 & 0 & & \dots & & 0 \\ \lambda_1 & 1 & 0 & & \dots & 0 \\ 0 & \lambda_2 & 1 & 0 & \dots & 0 \\ \vdots & & \ddots & \ddots & & \vdots \\ \vdots & & & \ddots & \ddots & 0 \\ 0 & 0 & & \dots & \lambda_k & 1 \end{bmatrix}, U_k = \begin{bmatrix} \zeta_1 & \beta_2 & 0 & \dots & & 0 \\ 0 & \zeta_2 & \beta_3 & 0 & \dots & 0 \\ 0 & & & & \dots & 0 \\ \vdots & & & \ddots & \ddots & \vdots \\ \vdots & & & & \ddots & \beta_k \\ 0 & 0 & & \dots & 0 & \zeta_k \end{bmatrix},$$

with $\zeta_1 = \alpha_1$, $\zeta_j = \alpha_j - \lambda_{j-1}\beta_j$ for $j = 2, 3, \dots, k$ and $\lambda_j = \frac{\beta_{j+1}}{\zeta_j}$ for $j = 1, 2, \dots, (k-1)$.

Since $T_k y_k = Q_k^T r^{(0)} = \beta_1 e_1$, where e_1 is the canonical unit vector,

$$\begin{aligned} x^{(k)} &= x^{(0)} + Q_k T_k^{-1} \beta_1 e_1 \\ &= x^{(0)} + Q_k (L_k U_k)^{-1} \beta_1 e_1 \\ &= x^{(0)} + Q_k U_k^{-1} L_k^{-1} \beta_1 e_1. \end{aligned}$$

Let $P_k = Q_k U_k^{-1}$ and $z_k = L_k^{-1}\beta_1 e_1$. Thus, from $P_k U_k = Q_k$, we can express the columns of P_k in terms of those of Q_k as $\zeta_j p_j + \beta_j p_{j-1} = q_j$, for every $1 \leq j \leq k$. We find P_j by iteratively calculating:

$$p_j = \frac{1}{\zeta_j}(q_j - \beta_j p_{j-1}).$$

Next, we construct the different components that make up the CG algorithm:

(1) Considering $P_k^T A P_k$ and substituting the definition of P_k gives:

$$\begin{aligned} P_k^T A P_k &= (Q_k U_k^{-1})^T A Q_k U_k^{-1} \\ &= U_k^{-T} Q_k^T A Q_k U_k^{-1} \\ &= U_k^{-T} T_k U_k^{-1} \\ &= U_k^{-T} L_k U_k U_k^{-1} \longrightarrow (\text{ from Cholesky}) \\ &= U_k^{-T} L_k. \end{aligned}$$

Since L_k is a lower triangular matrix, and U_k^{-T} is the transpose of an upper triangular matrix, and the product of two lower triangular matrices is lower triangular, $U_k^{-T}L_k$ is a lower triangular matrix. Thus, since $P_k^T AP_k$ is symmetric, it is diagonal, i.e, $p_i^T Ap_j = 0$ for all $i \neq j$. Since $A^T = A$, we get

$$(Ap_j, p_i) = (p_j, A^T p_i) = (p_j, Ap_i) = 0, \text{ for all } i \neq j. \tag{2.21}$$

The set of vectors $\{p_i\}_{i=1}^k$ are called direction vectors. From Equation (2.21), we see that these vectors are conjugate (orthogonal) to one another [21].

(2) Writing

$$\begin{aligned} x^{(k)} &= x^{(0)} + Q_k y_k && \text{from (*)} \\ b - Ax^{(k)} &= b - A(x^{(0)} + Q_k y_k) \\ b - Ax^{(k)} &= r^{(0)} - AQ_k y_k. \end{aligned}$$

Since $AQ_k = Q_{k+1}\hat{T}_k$ and $T_k y_k = Q_k^T r^{(0)}$, we get

$$\begin{aligned} r^{(k)} &= r^{(0)} - Q_k T_k y_k - \delta_k q_{k+1} \\ &= -\delta_k q_{k+1}, \end{aligned}$$

for some $\delta_k \in \mathbb{R}$. The last line shows $r^{(k)}$ to be in the same direction as q_{k+1}. Since the q's are orthogonal, the vectors in the set $\{r^{(i)}\}_{i=0}^k$ are orthogonal to one another, i.e, $(r^{(i)}, r^{(j)}) = 0$, for all $i \neq j$ [21]. Another interesting fact presented in [21] is that

$$(p_j, r^{(0)}) = (p_j, r^{(1)}) = (p_j, r^{(2)}) = \cdots = (p_j, r^{(i)}) = 0,$$

for $i < j$.

(3) From **(2)** and the relation $P_k U_k = Q_k$, we can write

$$p_{j+1} = r^{(j+1)} + \gamma_j p_j, \tag{2.22}$$

for $\gamma_j \in \mathbb{R}^k$. Multiplying $p_j^T A$ to both sides of (2.22) we get:

$$p_j^T A p_{j+1} = p_j^T A r^{(j+1)} + p_j^T A \gamma_j p_j,$$

and, using conjugate properties, $0 = p_j^T A r^{(j+1)} + p_j^T A \gamma_j p_j$. So,

$$\gamma_j = \frac{-p_j^T A r^{(j+1)}}{p_j^T A p_j} = \frac{-(Ar^{(j+1)}, p_j)}{(Ap_j, p_j)}. \tag{2.23}$$

(4) Let $x^{(j+1)} = x^{(j)} + \alpha_j p_j$. Using $r^{(j)} = b - Ax^{(j)}$,

$$\begin{aligned} Ax^{(j+1)} &= Ax^{(j)} + \alpha_j A p_j \\ b - Ax^{(j+1)} &= b - (Ax^{(j)} + \alpha_j A p_j). \end{aligned}$$

$$r^{(j+1)} = r^{(j)} - \alpha_j A p_j, \tag{2.24}$$

multiplying both sides with $(r^{(j)})^T$, we get $(r^{(j)})^T r^{(j+1)} = (r^{(j)})^T r^{(j)} - \alpha_j (r^{(j)})^T A p_j$. So, using $(r^{(j+1)}, r^{(j)}) = 0$, we have

$$\alpha_j = \frac{(r^{(j)})^T r^{(j)}}{(r^{(j)})^T A p_j} = \frac{(r^{(j)}, r^{(j)})}{(Ap_j, r^{(j)})}. \tag{2.25}$$

By (2.22), we see that $r^{(j+1)} = p_{j+1} - \gamma_j p_j$ and $(r^{(j)})^T = (p_j - \gamma_{j-1} p_{j-1})^T$. So,

$$\begin{aligned} (r^{(j)})^T A p_j &= (p_j - \gamma_{j-1} p_{j-1})^T A p_j \\ &= p_j^T A p_j - \gamma_j p_{j-1}^T A p_j \\ &= p_j^T A p_j. \end{aligned}$$

Thus, (2.25) becomes

$$\alpha_j = \frac{(r^{(j)}, r^{(j)})}{(Ap_j, p_j)}. \tag{2.26}$$

(5) Rearranging (2.24) to $Ap_j = \frac{1}{\alpha_j}(r^{(j)} - r^{(j+1)})$, we can substitute this in (2.23) to give:

$$\begin{aligned}\gamma_j &= \frac{-p_j A r^{(j+1)}}{p_j^T A p_j} \\ &= \frac{-(Ap_j)^T r^{(j+1)}}{p_j^T A p_j} \\ &= \frac{(r^{(j+1)}-r^{(j)})^T}{\alpha_j}.\frac{r^{(j+1)}}{p_j^T A p_j} \\ &= \frac{1}{\alpha_j}.\frac{(r^{(j+1)})^T r^{(j+1)}}{p_j^T A p_j}.\end{aligned}$$

From (2.26), $\alpha_j p_j^T A p_j = (r^{(j)})^T r^{(j)}$. So,

$$\gamma_j = \frac{(r^{(j+1)})^T r^{(j+1)}}{(r^{(j)})^T r^{(j)}} = \frac{(r^{(j+1)}, r^{(j+1)})}{(r^{(j)}, r^{(j)})}.$$

These five steps lead to the common form of the Conjugate Gradient (CG) algorithm [46, 30].

CG algorithm:

Given $x^{(0)}$, $r^{(0)} = b - Ax^{(0)}$, $p_0 = r^{(0)}$:
For $j = 0, 1, 2, \ldots$
$\{\alpha_j = \frac{(r^{(j)}, r^{(j)})}{(Ap_j, p_j)}$
$x^{(j+1)} = x^{(j)} + \alpha_j p_j$
$r^{(j+1)} = r^{(j)} - \alpha_j A p_j$
$\gamma_j = \frac{(r^{(j+1)}, r^{(j+1)})}{(r^{(j)}, r^{(j)})}$
$p_{j+1} = r^{(j+1)} + \gamma_j p_j$.
$\}$

Since $x^{(j)}$, $r^{(j)}$, and p_j are needed to compute $x^{(j+1)}$, $r^{(j+1)}$, and p_{j+1}, usually the method stores 3 vectors: x, r, and p_{j+1} [21, 46, 36].

Cost: Computation wise, the cost of CG is fairly small compared with that of previous methods. It requires only 1 matrix-vector product, 2 inner products, and 3 vector updates per iteration [21].

Convergence:
In exact arithmetic, the method converges in at most n iterations, so the method can

not only be considered as a Krylov method, but also a direct one [21].

The general bound on the rate of convergence of CG is

$$||x - x^{(j+1)}||_A \leq 2||x - x^{(0)}||_A \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^{j+1}, \tag{2.27}$$

where $x^{(0)}$, $\{x^{(j)}\}_{j=1}^{n}$ are the initial guess and set of solution vectors that are obtained from the CG algorithm, respectively [43]. $\kappa(A)$ is called the condition number of matrix A, and is the ratio of the maximum eigenvalue of A to the minimum, $\kappa(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$. Note, $||.||_A^2$ is called the A-norm, where $||s||_A^2 = (s, As) = s^T As$. The best possible $x^{(i+1)}$ is that which makes $x - x^{(i+1)}$, the error at the $(i+1)^{th}$ iteration, to be the smallest possible. From (2.18), we see that this, in general, happens fastest when $\kappa(A)$ is as close to 1 as possible, leading to faster CG convergence.

Preconditioned Conjugate Gradient:

CG, like most other Krylov methods, works more efficiently when transforming the system $Ax = b$ into $M^{-1}Ax = M^{-1}b$, where M^{-1} is called a preconditioning matrix (or preconditioner). Preconditioned systems are generally easier to solve than their original counterparts for sensible choices of the preconditioner. That is why preconditioning methods have been an extensive focus for research in the past few decades [25, 43, 11]. Equivalent to the system $M^{-1}Ax = M^{-1}b$ is $M^{-\frac{1}{2}}AM^{-\frac{1}{2}}M^{\frac{1}{2}}x = M^{-\frac{1}{2}}b$. To set notation, we shall apply the CG method to

$$\hat{A}\hat{x} = \hat{b} \tag{2.28}$$

to derive the Preconditioned Conjugate Gradient (PCG), where $\hat{A} = M^{-\frac{1}{2}}AM^{-\frac{1}{2}}$, $\hat{x} = M^{\frac{1}{2}}x$ and $\hat{b} = M^{-\frac{1}{2}}b$. We assume that M^{-1} is also an SPD matrix, so that its principle square root, $M^{-\frac{1}{2}}$, is well-defined, and $\hat{A}$ is SPD if A is.

We note the following notation to be used, following [43]

$$\begin{aligned}
\hat{x}^{(j)} &= M^{\frac{1}{2}} x^{(j)}, \\
\hat{r}^{(j)} &= M^{-\frac{1}{2}} r^{(j)}, \\
\hat{p}_j &= M^{\frac{1}{2}} p_j, \\
w_j &= A p_j, \\
\hat{w}_j &= M^{\frac{1}{2}} w_j,
\end{aligned}$$

where $x^{(j)}$,$r^{(j)}$ and p_j are as above.

(1) From $\hat{r}^{(0)} = \hat{b} - \hat{A}\hat{x}^{(0)}$, we have

$$\begin{aligned}
M^{-\frac{1}{2}} r^{(0)} &= M^{-\frac{1}{2}} b - M^{-\frac{1}{2}} A M^{-\frac{1}{2}} M^{\frac{1}{2}} x^{(0)} \\
&= M^{-\frac{1}{2}} [b - A x^{(0)}].
\end{aligned}$$

(2) At $j = 0$, p_0 and $\hat{p}_0$ are found by

$$\begin{aligned}
\hat{p}_0 &= \hat{r}^{(0)} \\
M^{\frac{1}{2}} p_0 &= M^{-\frac{1}{2}} r^{(0)} \\
p_0 &= M^{-\frac{1}{2}} M^{-\frac{1}{2}} r^{(0)} \\
&= M^{-1} r^{(0)}.
\end{aligned}$$

Since M is SPD, $(M^{-\frac{1}{2}})^T = M^{-\frac{1}{2}}$. Let $y_0 = M^{-1} r^{(0)}$, i.e $M y_0 = r^{(0)}$. So,

$$p_0 = y_0.$$

(3) Taking $y_j = M^{-1} r^{(j)}$, α_j becomes

$$\begin{aligned}
\alpha_j &= \frac{(\hat{r}^{(j)})^T \hat{r}^{(j)}}{\hat{p_j}^T \hat{A} \hat{p_j}} \\
&= \frac{(M^{-\frac{1}{2}} r^{(j)})^T M^{-\frac{1}{2}} r^{(j)}}{(M^{\frac{1}{2}} p_j)^T (M^{-\frac{1}{2}} A M^{-\frac{1}{2}})(M^{\frac{1}{2}} p_j)} \\
&= \frac{(r^{(j)})^T M^{-\frac{1}{2}} M^{-\frac{1}{2}} r^{(j)}}{p_j^T M^{\frac{1}{2}} M^{-\frac{1}{2}} A M^{-\frac{1}{2}} M^{\frac{1}{2}} p_j} \\
&= \frac{(r^{(j)})^T M^{-1} r^{(j)}}{p_j^T A p_j} \\
&= \frac{(r^{(j)})^T y_j}{p_j^T A p_j} \\
&= \frac{(r^{(j)}, y_j)}{(p_j, A p_j)}.
\end{aligned}$$

(4) For $x^{(j+1)}$,

$$\begin{aligned}
\hat{x}^{(j+1)} &= \hat{x}^{(j)} + \alpha_j \hat{p}_j \\
M^{\frac{1}{2}} x^{(j+1)} &= M^{\frac{1}{2}} x^{(j)} + \alpha_j M^{\frac{1}{2}} p_j \\
M^{\frac{1}{2}} x^{(j+1)} &= M^{\frac{1}{2}} [x^{(j)} + \alpha_j p_j] \\
x^{(j+1)} &= x^{(j)} + \alpha_j p_j.
\end{aligned}$$

(5) $r^{(j+1)}$ becomes, from $\hat{r}^{(j+1)}$,

$$\begin{aligned}
\hat{r}^{(j+1)} &= \hat{r}^{(j)} - \alpha_j \hat{A}\hat{p}_j \\
M^{-\frac{1}{2}} r^{(j+1)} &= M^{-\frac{1}{2}} r^{(j)} - \alpha_j M^{-\frac{1}{2}} A M^{-\frac{1}{2}} M^{\frac{1}{2}} p_j \\
M^{-\frac{1}{2}} r^{(j+1)} &= M^{-\frac{1}{2}} [r^{(j)} - \alpha_j A p_j] \\
r^{(j+1)} &= r^{(j)} - \alpha_j A p_j.
\end{aligned}$$

(6) To find β_j

$$\begin{aligned}
\beta_j &= \frac{(\hat{r}^{(j+1)})^T \hat{r}^{(j+1)}}{(\hat{r}^{(j)})^T \hat{r^{(j)}}} \\
&= \frac{(r^{(j+1)})^T M^{-\frac{1}{2}} M^{-\frac{1}{2}} r^{(j+1)}}{(r^{(j)})^T M^{-\frac{1}{2}} M^{-\frac{1}{2}} r^{(j)}} \\
&= \frac{(r^{(j+1)})^T M^{-1} r^{(j+1)}}{(r^{(j)})^T M^{-1} r^{(j)}} \\
&= \frac{(r^{(j+1)})^T y_{j+1}}{(r^{(j)})^T y_j} \\
&= \frac{(r^{(j+1)}, y_{j+1})}{(r^{(j)}, y_j)}.
\end{aligned}$$

(7) For p_{j+1},

$$\begin{aligned}
\hat{p}_{j+1} &= \hat{r}^{(j+1)} + \beta_j \hat{p}_j \\
M^{\frac{1}{2}} p_{j+1} &= M^{-\frac{1}{2}} r^{(j+1)} + \beta_j M^{\frac{1}{2}} p_j \\
p_{j+1} &= M^{-\frac{1}{2}} [M^{-\frac{1}{2}} r^{(j+1)} + \beta_j M^{\frac{1}{2}} p_j] \\
&= M^{-1} r^{(j+1)} + \beta_j p_j \\
&= y_{j+1} + \beta_j p_j.
\end{aligned}$$

Now the PCG algorithm becomes [43]

PCG algorithm:

Let $x^{(0)}$ be initial guess, $r^{(0)} = b - Ax^{(0)}$, $p_0 = y_0$. Solve $My_0 = r^{(0)}$,

For $j = 0, 1, \ldots$

$\{\ \alpha_j = \frac{(r^{(j)}, y_i)}{(p_j, Ap_j)}$

$x^{(j+1)} = x^{(j)} + \alpha_j p_j$
$r^{(j+1)} = r^{(j)} - \alpha_j A p_j$
Solve $My_{j+1} = r^{(j+1)}$
$\beta_j = \frac{(r^{(j+1)}, y_{j+1})}{(r^{(j)}, y_j)}$
$p_{j+1} = y_{j+1} + \beta_j p_j$.
}

Cost: In terms of storage, PCG needs to store 4 vectors, one more than unpreconditioned CG, and 2 matrices, A and M (although these may be only implicitly stored). As for computation, it has almost the same cost as CG, but with the addition of an extra system solve per iteration, in the step requiring solution of $My_{j+1} = r^{(j+1)}$.

Due to this extra step, M must be easy to assemble, and it should be cheap to solve $My_{j+1} = r^{(j+1)}$, or else the goal of increasing effeciency of CG with PCG will not be achieved. There are many choices of preconditioners from which one can choose. Details on incomplete Cholesky preconditioners, incomplete block preconditioners and others, are presented in [16]. The simplest choice for a preconditioner is the Jacobi (or diagonal scaling) preconditioner, with $M = diag(A)$. To gain some insight into the choice of M, consider the 2 extreme cases of either $M = I$ or $M = A$. In the first case, $M = I$ only gets us back to the CG method, so there is not much use in applying this preconditioner. It is cheap to use, but does not improve performance. As for $M = A$, this gives $M^{-1}Ax = M^{-1}b \longrightarrow A^{-1}Ax = A^{-1}b$, which is again the original complicated problem $x = A^{-1}b$. Thus, this is effective, but has an impractical cost per iteration. So, the choice of M is usually, in a sense, between these two extremes.

Other factors also affect the performance of PCG, like the choice of initial guess and the stopping criteria. They are omitted from our discussion, but can be found in Chapter 2 of [43].

Convergence: Again we derive the convergence rate from that of CG:

Theorem 4 *Given initial guess $x^{(0)}$, after $j+1$ steps of PCG, the approximate solution satisfies*

$$||x - x^{(j+1)}||_A \leq 2||x - x^{(0)}||_A \left(\frac{\sqrt{\kappa(M^{-1}A)} - 1}{\sqrt{\kappa(M^{-1}A)} + 1} \right)^{j+1},$$

where $\kappa(M^{-1}A) = \frac{\lambda_{\max}(M^{-\frac{1}{2}}AM^{-\frac{1}{2}})}{\lambda_{\min}(M^{-\frac{1}{2}}AM^{-\frac{1}{2}})}$.

Proof

From the convergence rate of CG with the system $\hat{A}\hat{x} = \hat{b}$, we have

$$||\hat{x} - \hat{x}^{(j+1)}||_{\hat{A}} \leq 2||\hat{x} - \hat{x}^{(0)}||_{\hat{A}} \left(\frac{\sqrt{\kappa(\hat{A})} - 1}{\sqrt{\kappa(\hat{A})} + 1} \right)^{j+1}.$$

Since $||s||_A^2 = (s, As) = s^T As$,

$$\begin{aligned}
||\hat{x} - \hat{x}^{(j+1)}||_{\hat{A}}^2 &= (\hat{x} - \hat{x}^{(j+1)})^T \hat{A} (\hat{x} - \hat{x}^{(j+1)}) \\
&= (M^{\frac{1}{2}}x - M^{\frac{1}{2}}x^{(j+1)})^T M^{-\frac{1}{2}} A M^{-\frac{1}{2}} (M^{\frac{1}{2}}x - M^{\frac{1}{2}}x^{(j+1)}) \\
&= (M^{\frac{1}{2}}[x - x^{(j+1)}])^T M^{-\frac{1}{2}} A M^{-\frac{1}{2}} (M^{\frac{1}{2}}[x - x^{(j+1)}]) \\
&= (x - x^{(j+1)})^T M^{\frac{1}{2}} M^{-\frac{1}{2}} A M^{-\frac{1}{2}} M^{\frac{1}{2}} (x - x^{(j+1)}) \\
&= (x - x^{(j+1)})^T A (x - x^{(j+1)}) \\
&= ||x - x^{(j+1)}||_A^2
\end{aligned}$$

Using this, and the fact that $\hat{A} = M^{-\frac{1}{2}}AM^{-\frac{1}{2}}$ is similar to $M^{-1}A$, we have our result.

□

Deflation:

We have seen the natural bound on the convergence rate of PCG depends on the maximum and minimum eigenvalues of the matrix $M^{-1}A$. Typically, the smallest eigenvalues correspond to the slowest to converge components of the solution to the system [45]. In order to eliminate the effects of these eigenvalues on the convergence of PCG, we can wisely choose a preconditioner that specifically deals with these modes

to enhance the overall efficiency of the PCG method. This is what leads to the introduction of deflation and the deflation preconditioner.

Following [43, 25] in our presentation of the deflation method, we begin with the following definitions

$$\begin{aligned} E &= Z^T A Z, \\ Q &= Z E^{-1} Z^T, \\ P &= I - AQ. \end{aligned}$$

Here, Z is a given $n \times k$ matrix, $(k < n)$, with full rank, meaning that its columns are linearly independent. E is called a Galerkin matrix or coarse matrix, Q is called the correction matrix, and P is the deflation matrix. The columns of Z are the deflation vectors, sometimes called projection vectors. It is easy to see that E is invertible, when A is SPD. E and Q are both symmetric, and P is a projection; meaning $P^2 = P$. Some interesting properties of E, Q, and P can be found in [43]. A proper choice of Z is key to achieving good performance from the deflated PCG algorithm. The definition of Z is treated in a set-up phase that occurs before the solve phase. The algorithms presented below are for the solve phase alone; in Chapter 3, we discuss the set-up phase where Z is defined.

Deflated CG:

Given P, we make the following decomposition of x:

$$x = (I - P^T)x + P^T x. \tag{2.29}$$

$(I - P^T)x = Qb$ and, so, this is easy to compute directly without knowing x. The difficulty lies in computing $P^T x$. This can be resolved by solving the system

$$PA\hat{x} = Pb, \tag{2.30}$$

where $\hat{x} = x + y$, such that x is the solution of $Ax = b$ and y belongs to the null space of P^T [40]. $\hat{x}$ is called the deflated solution, and (2.30) is referred to as the deflated

system. This can be observed as a preconditioned system with the deflation matrix, P, as the preconditioner.

After solving for $\hat{x}$, we then have the equality

$$P^T\hat{x} = P^T(x + y) = P^T x,$$

since $P^T y = 0$. So, (2.29) becomes $x = Qb + P^T\hat{x}$.

This deflated Conjugate Gradient algorithm is as follows.

Deflated CG algorithm: [43]

Let $x^{(0)}$ be initial guess, $r^{(0)} = b - Ax^{(0)}$, $p_0 = \hat{r}^{(0)}$, and $\hat{r}^{(0)} = Pr^{(0)}$,
For $j = 0, 1, \ldots$
$\{$ $\hat{w}_j = PAp_j$
$\alpha_j = \frac{(\hat{r}^{(j)}, \hat{r}^{(j)})}{(\hat{w}_j, p_j)}$
$\hat{x}^{(j+1)} = \hat{x}_j + \alpha_j p_j$
$\hat{r}^{(j+1)} = \hat{r}^{(j)} - \alpha_j \hat{w}_j$
$\beta_j = \frac{(\hat{r}^{(j+1)}, \hat{r}^{(j+1)})}{(\hat{r}^{(j)}, \hat{r}^{(j)})}$
$p_{j+1} = \hat{r}^{(j+1)} + \beta_j p_j.$
$\}$

The deflated CG method is an extension of the CG method, and all components in deflated CG marked with the "hat" notation are called deflated components. So, $\hat{r}$ is called the deflated residual, since it is the residual of (2.30).

Deflated Preconditioned Conjugate Gradient method:

Deflated CG often suffers from similar poor performance as unpreconditioned CG, unless the deflation space is prohibitively large. To improve performance, we can also

use a preconditioner within deflated CG, effectively adding a preconditioning operation to (2.30). This gives a method referred to as the deflated PCG algorithm. This method is often seen to be more robust than PCG and more stable [25].

Applying a preconditioner, (2.30) becomes

$$M^{-1}PA\hat{x} = M^{-1}Pb,$$

which is equivalent to

$$\overline{P}(M^{-\frac{1}{2}}AM^{-\frac{1}{2}})(M^{\frac{1}{2}}x) = \overline{P}(M^{-\frac{1}{2}}b)$$

or

$$\overline{P}\overline{A}\bar{x} = \overline{P}\bar{b},$$

where $\overline{A} = M^{-\frac{1}{2}}AM^{-\frac{1}{2}}$, $\bar{x} = M^{\frac{1}{2}}$, $\bar{b} = M^{-\frac{1}{2}}b$, [43], and the definitions from before become

$$\begin{aligned} \overline{E} &= \overline{Z}^T\overline{A}\overline{Z} \\ \overline{Q} &= \overline{Z}\overline{E}^{-1}\overline{Z}^T \\ \overline{P} &= I - \overline{A}\overline{Q}. \end{aligned}$$

Here, $\overline{Z}$ is a preconditioned deflation matrix; easy calculations show that $\overline{P} = M^{-\frac{1}{2}}PM^{-\frac{1}{2}}$ when $Z = M^{-\frac{1}{2}}\overline{Z}$.

In the following, the deflated preconditioned conjugate gradient algorithm is presented.

Deflated PCG algorithm: [43]

Let $x^{(0)}$ be initial guess, $r^{(0)} = b - Ax^{(0)}$,$\hat{r}^{(0)} = Pr^{(0)}$. Solve $My_0 = \hat{r}^{(0)}$, $p_0 = y_0$,
For $j = 0, 1, \ldots$
$\{$ $\hat{w}_j = PAp_j$
$\alpha_j = \frac{(\hat{r}^{(j)}, y_i)}{(p_j, \hat{w}_j)}$
$\hat{x}^{(j+1)} = \hat{x}^{(j)} + \alpha_j p_j$

$\hat{r}^{(j+1)} = \hat{r}^{(j)} - \alpha_j \hat{w}_j$
Solve $My_{j+1} = \hat{r}^{(j+1)}$
$\beta_j = \frac{(\hat{r}^{(j+1)}, y_{j+1})}{(\hat{r}^{(j)}, y_j)}$
$p_{j+1} = y_{j+1} + \beta_j p_j.$
}

This can be derived in a similar manner as we derived PCG.

Theorem 5 *Let A, M, and Z be given, with A and M SPD $n \times n$ matrices, and Z be a full rank $n \times k$ matrix, with $k < n$. Let $P = I - AZ(Z^T AZ)^{-1}Z^T$ and let $r^{(0)} = P(b - Ax^{(0)})$, for given right-hand side vector, b, and initial guess, $x^{(0)}$. Then, the PCG algorithm with balancing preconditioner $P^T M^{-1} P$ is equivalent to the deflated PCG algorithm with preconditioner $M^{-1}P$ up to a shift in the null space of P^T, Range(Z).*

Proof: We prove this by induction.

(1) Let $j = 0$:

In PCG: Let $p_0 = y_0 = P^T M^{-1} P r^{(0)}$.

(a) $\omega_0 = Ap_0$
$= PAM^{-1}Pr^{(0)}.$

(b) $\alpha_0 = \frac{y_0^T r^{(0)}}{\omega_0^T p_0}$
$= \frac{(r^{(0)})^T P^T M^{-1} P r^{(0)}}{(r^{(0)})^T P^T M^{-1} A P^T p_0}.$

In DPCG: Let $p_0 = y_0 = M^{-1}Pr^{(0)}$.

(a) $\hat{\omega_0} = PAp_0$
$= PAM^{-1}Pr^{(0)}.$

(b) $\alpha_0 = \frac{y_0^T \hat{r}^{(0)}}{\hat{\omega}_0^T p_0}$
$= \frac{(r^{(0)})^T P^T M^{-1} P r^{(0)}}{(r^{(0)})^T P^T M^{-1} A P^T p_0}.$

So, in both algorithms the steps correspond to one another for $j = 0$.

(2) Now we prove for general case j. Assume the two algorithms are equivalent for $j-1$,

so that

$$\hat{r}^{(j)} = r^{(j)},\ \ y_{j-1} = P^T\hat{y}_{j-1},\ \ \beta_{j-1} = \hat{\beta}_{j-1}, \text{ and } \ p_{j-1} = P^T\hat{p}_{j-1}.$$

For the PCG algorithm, using the definitions given above, we have the set-up phase equivalent, so

(c) $r^{(j)} = r^{(j-1)} - \alpha_{j-1}Ap_{j-1}$

$$= Pr^{(j-1)} - \frac{(r^{(j-1)})^T P^T M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p^{(j-1)}} (r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}.$$

(d) $y_j = P^T M^{-1} P r^{(j)}$

$$= P^T M^{-1} P r^{(j-1)} - P^T M^{-1} \frac{(r^{(j-1)})^T P^T M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p^{(j-1)}} (r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}.$$

(e) $\beta_{j-1} = \frac{y_j^T r^{(j)}}{y_{j-1}^T r^{(j)}}$

$$= [1 - \frac{(r^{(j-1)})^T P^T M^{-1} P A M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}} - \frac{(r^{(j-1)})^T P^T M^{-1} A P^T M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}} +$$
$$+ \frac{(r^{(j-1)})^T P^T M^{-1} A P^T M^{-1} (r^{(j-1)})^T P^T M^{-1} P r^{(j-1)} P A M^{-1} P r^{(j-1)}}{((r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1})^2}].$$

(f) $p_j = \beta_{j-1} p_{j-1} - y_j$

$$= P^T[\beta_{j-1} M^{-1} P r^{(j-1)} + M^{-1} P r^{(j-1)}$$
$$- M^{-1} \frac{(r^{(j-1)})^T P^T M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p^{(j-1)}} (r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}].$$

For DPCG:

(c) $\hat{r}^{(j)} = \hat{r}^{(j-1)} - \alpha_{j-1}\hat{\omega}_{j-1}$

$$= Pr^{(j-1)} - \frac{(r^{(j-1)})^T P^T M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p^{(j-1)}} (r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}.$$

(d) $y_j = M^{-1}\hat{r}^{(j)}$

$= M^{-1}Pr^{(j-1)} - M^{-1}\frac{(r^{(j-1)})^T P^T M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p^{(j-1)}} (r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}.$

(e) $\beta_{j-1} = \frac{(\hat{r}^{(j)})^T M^{-1} \hat{r}^{(j)}}{y_{j-1}^T \hat{r}^{(j-1)}}$

$$= [1 - \frac{(r^{(j-1)})^T P^T M^{-1} P A M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}} - \frac{(r^{(j-1)})^T P^T M^{-1} A P^T M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}} + \frac{(r^{(j-1)})^T P^T M^{-1} A P^T M^{-1} (r^{(j-1)})^T P^T M^{-1} P r^{(j-1)} P A M^{-1} P r^{(j-1)}}{((r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1})^2}].$$

(f) $p_j = \beta_{j-1} p_{j-1} - y_{j-1}$

$$= P^T [\beta_{j-1} M^{-1} P r^{(j-1)} + M^{-1} P r^{(j-1)} - \ldots \ldots M^{-1} \frac{(r^{(j-1)})^T P^T M^{-1} P r^{(j-1)}}{(r^{(j-1)})^T P^T M^{-1} A P^T p^{(j-1)}} (r^{(j-1)})^T P^T M^{-1} A P^T p_{j-1}].$$

This concludes that the algorithms are equivalent up to a shift in the direction of the null space of P^T.

□

The error bound of the deflated PCG algorithm is now

$$||x - x^{(j+1)}||_A \leq 2||x - x^{(0)}||_A \left(\frac{\sqrt{\tilde{\kappa}(M^{-1}PA)} - 1}{\sqrt{\tilde{\kappa}(M^{-1}PA)} + 1} \right)^{j+1},$$

where $\tilde{\kappa}(S)$ is called the effective condition number and is the ratio of the largest eigenvalue to the smallest non-zero eigenvalue of a SPD matrix S, i.e., $\tilde{\kappa}(S) = \frac{\lambda_{max}(S)}{\min_{\lambda \neq 0} \lambda(S)}$.

The smaller $\tilde{\kappa}(M^{-1}PA)$ is, the faster convergence of deflated PCG becomes, i.e. the more accurate the approximation $x^{(j+1)}$ will be in comparision with the exact solution, x. To make effective use of this idea, deflation vectors can be chosen, in such a way as to reduce the value of $\kappa(M^{-1}PA)$, such as if eigenvectors of $M^{-1}A$ corresponding to extreme eigenvalues were used for deflation vectors then $\tilde{\kappa}(M^{-1}PA)$ is decreased [43]. We also see in [43] that $\kappa(M^{-1}A) > \tilde{\kappa}(M^{-1}PA)$, which makes using deflation potentially an effective method to improve performance of the PCG method.

The efficiency of deflated methods depends strongly on our choice of the deflation vectors in Z. To acheive our goal of using deflation, must choose the columns of Z wisely to get faster convergence. Usually, the best choice of Z varies depending on the problem at hand. Many techniques have been suggested for the choice of these vectors [43, 25]. An impractical choice would be to choose eigenvectors of the system $M^{-1}A$ to reduce the condition number $\tilde{\kappa}(M^{-1}PA)$. Typically, we try to make choices to either improve the clustering of eigenvalues of the deflated system or to reduce its effective condition number, both of which lead to improved performance. One choice is to take the domain of the problem and divide it into subdomains. For each subdomain, we take a column of Z with entries of 1 for the nodes in that subdomain and 0 for all other points outside. In this case, called subdomain deflation, Z is a matrix of zeros and ones. Another possibility is to build vectors of Z from eigenvectors of the system, similar to what we will consider later on in our research. A recycling delation method could be also used [43] or multigrid/ multilevel deflation [11]. For a more detailed discussion of these methods, see [43].

As an additional note, the preconditioner defined in Theorem 5 for PCG is called the balancing preconditioner due to its construction using P. Since we have established through this theorem that both PCG and DCG are equivalent with the certain set-up of preconditioners and because standard software packages, such as PETSc, offer well-tested and optimized versions of standard PCG, but not DPCG, we implement the deflation preconditioners described later as balancing preconditioners for standard PCG. This is discussed in more detail in Chapter 3.

2.3 Multigrid methods

Differential equations can naturally be approximated using different sized grids. This is used as an advantage in a class of numerical methods called multigrid methods, first developed in the 1960's [7, 4, 35, 44, 6]. These methods use multiple levels of division of the domain to efficiently approximate the solution of a boundary value problem. This, thus, contributes to solving a bigger range of problems more efficiently than

previous numerical methods. In fact, they are among the fastest numerical solution techniques developed to this day for a wide range of discretized elliptic differential equations. We tend to use them because they are faster to converge than classical iterative methods, although they are still iterative methods. This class of method includes diverse techniques in the way the levels of the multigrid hierarchy constructed, either using geometric information obtained from the problem or algebraically constructing them using properties obtained only from the matrices defining the system of equations at hand. The first group of approaches are called geometric multigrid and the latter are called algebraic multigrid. Both are presented in the following subsection. Many other types of multigrid methods exist, including adaptive methods and "black-box" multigrid methods. For more details on these, see [12, 7, 44].

We will consider the following boundary value problem,

$$\begin{aligned} -u''(x) &= f, \qquad \text{in } \Omega = (0,1), \\ u(0) &= u(1) = 0, \end{aligned} \tag{2.31}$$

as our model problem and aim to solve (2.31) using multigrid methods. These numerical methods are used to solve the linear system of equations formed by the discretization of partial differential equations. Thus, we introduce finite-difference and finite-element discretizations. Finite differences will be discussed here and finite elements in Chapter 3.

First, we divide Ω into a set of points $\{x_i\}_{i=0}^{n}$ such that

$$0 = x_0 < x_1 < \cdots < x_{n-1} < x_n = 1.$$

To generate a uniform mesh, we take $h = \frac{1}{n}$ and $x_i = ih$; see Figure 2.1. h is referred to as the mesh size.

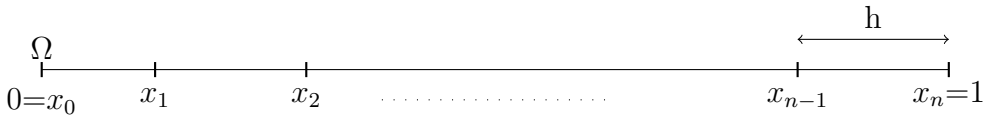


Figure 2.1: Uniform mesh with nodes $\{0, x_1, \ldots, x_{n-1}, 1\}$

.

Finite differences:

Recall the definition of the derivative of a smooth function, u, at a point $x \in \mathbb{R}$ is

$$u'(x) = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h}.$$

We clearly can approximate $u'(x)$ by replacing the limit by a fixed (small) value of h. Doing this when h is as close to zero as possible gives the best possible approximation of $u'(x)$.

Consider the Taylor expansion with Lagrange remainder. Assuming $u \in C^{n+1}(0,1)$, we have

$$u(x+h) = u(x) + hu'(x) + \frac{h^2}{2!}u''(x) + \frac{h^3}{3!}u'''(x) + \cdots + \frac{h^n}{n!}u^{(n)}(x) + \frac{h^{n+1}}{(n+1)!}u^{(n+1)}(\xi), \tag{2.32}$$

where $u^{(n)}(x)$ is the n^{th} derivative of u, h^n is the n^{th} power of h, and ξ is a real number between x and $x+h$. Taking $n=1$ yields $u(x+h) = u(x) + hu'(x) + O(h^2)$, where $O(h^2)$ includes the remainder terms of the expansion, again gives the approximation

$$u'(x) \approx \frac{u(x+h) - u(x)}{h}.$$

We can also write out the derivative exactly as

$$u'(x) = \frac{u(x+h) - u(x)}{h} - \frac{h}{2!}u''(\xi),$$

where the $O(h^2)$ term neglected in the approximation above is included in Lagrange form as $-\frac{h^2}{2!}u''(\xi)$, scaled by $\frac{1}{h}$.

Using our mesh notation above, we write u at mesh points x_{i+1} and x_{i-1} as

$$\begin{aligned} u(x_{i+1}) = &\quad u(x_i) + (x_{i+1} - x_i)u'(x_i) + \frac{(x_{i+1} - x_i)^2}{2!}u''(\xi_+), \\ u(x_{i-1}) = &\quad u(x_i) + (x_{i-1} - x_i)u'(x_i) + \frac{(x_{i-1} - x_i)^2}{2!}u''(\xi_-), \end{aligned} \tag{2.33}$$

where $\xi_+ \in (x_i, x_{i+1})$ and $\xi_- \in (x_{i-1}, x_i)$. Denoting $h_i = x_i - x_{i-1}$, we get:

$$u(x_{i+1}) = u(x_i) + h_{i+1}u'(x_i) + \frac{h_{i+1}^2}{2!}u''(\xi_+) \tag{2.34a}$$

$$u(x_{i-1}) = u(x_i) - h_i u'(x_i) + \frac{h_i^2}{2!}u''(\xi_-). \tag{2.34b}$$

There are three kinds of difference methods that can be directly obtained from (2.34): forward, backward and central differences.

The *forward difference* approximation is taken from (2.34a) :

$$u'(x_i) = \frac{u(x_{i+1}) - u(x_i)}{h_{i+1}} - \frac{h_{i+1}}{2!}u''(\xi_+).$$

The *backward difference* approximation is taken from (2.34b) :

$$u'(x_i) = \frac{-u(x_i) + u(x_{i-1})}{h_i} + \frac{h_i}{2!}u''(\xi_-).$$

The *central difference* approximation is the weighted average of the last two differences:

$$\begin{aligned} u'(x_i) = \frac{h_i}{h_i + h_{i+1}}\Big(\frac{u(x_{i+1}) - u(x_i)}{h_{i+1}} - \frac{h_{i+1}}{2}u''(\xi_+)\Big) + \\ + \frac{h_{i+1}}{h_i + h_{i+1}}\Big(\frac{u(x_i) - u(x_{i-1})}{h_i} + \frac{h_i}{2}u''(\xi_-)\Big). \end{aligned} \tag{2.35}$$

The central difference approximation is important for getting a more precise approximation of the derivative by taking more terms in the expansion. For example,

writing

$$\begin{aligned} u(x_{i+1}) &= u(x_i) + h_{i+1}u'(x_i) + \frac{h_{i+1}^2}{2!}u''(x_i) + \frac{h_i^3}{3!}u'''(\xi_+) \\ u(x_{i-1}) &= u(x_i) - h_i u'(x_i) + \frac{h_i^2}{2!}u''(x_i) - \frac{h_i^3}{3!}u'''(\xi_-), \end{aligned} \tag{2.36}$$

and considering the uniform-mesh case with $h = h_{i+1} = h_i$ gives us a central difference approximation,

$$u'(x_i) = \frac{u(x_{i+1}) - u(x_{i-1})}{2h} - \frac{h^2}{12}\Big(u'''(\xi_+) + u'''(\xi_-)\Big).$$

If $u \in C^3((0,1))$, this gives an $O(h^2)$ approximation as opposed to the $O(h)$ precision we get for the forward and backward differences.

By the exact same process, we can find expressions for difference equations approximating higher derivatives by simply isolating the higher derivative on the left-hand side in place of $u'(x_i)$. For the second derivative on a uniform mesh, the central difference scheme is

$$u''(x_i) = \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2} - \frac{h^2}{24}\Big(u''''(\xi_+) + u''''(\xi_-)\Big).$$

Approximations to partial derivatives on higher dimensional meshes can also be found similarly.

Geometric Multigrid:

Going back to our model problem, we approximate $u_i \approx u(x_i)$ and write $f_i = f(x_i)$, giving the central difference approximation to $-u'' = f$ as

$$\begin{aligned} \frac{1}{h^2}\Big(- u_{i+1} + 2u_i - u_{i-1}\Big) &= f_i, \qquad\qquad \text{for } 1 \le i \le n-1 \\ u_0 = u_n &= 0. \end{aligned} \tag{2.37}$$

This leads to a linear system of equations

$$Au = f, \tag{2.38}$$

where

$$A = \begin{bmatrix} \frac{2}{h^2} & \frac{-1}{h^2} & 0 & & \dots & 0 \\ \frac{-1}{h^2} & \frac{2}{h^2} & \frac{-1}{h^2} & 0 & \dots & 0 \\ \vdots & \vdots & & & & \vdots \\ & & & & & \vdots \\ 0 & \dots & \dots & 0 & \frac{-1}{h^2} & \frac{2}{h^2} \end{bmatrix}, u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{n-1} \end{bmatrix}, f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ f_{n-1} \end{bmatrix}. \tag{2.39}$$

Notice that A is a symmetric, positive-definite, sparse, and diagonally dominant matrix. We denote v as an approximation vector to the exact solution u, with $v = (v_1, \dots, v_{n-1})^T$. As in previous sections, the error is denoted as $e = u - v$ and the residual is $r = f - Av = Ae$.

The idea of residual correction is to solve $Ae = r$ to get a correction to the approximation, v [7]. This idea plays a big role in multigrid methods. The solution of system (2.37) can be approximated using a relaxation scheme like weighted Jacobi or GS.

Using a standard ansatz, we let

$$V_i^{(k)} = \sin\left(\frac{k\pi i}{n}\right) \tag{2.40}$$

be the i^{th} component of the k^{th} eigenvector of A in (2.38) for $1 \leq k \leq n-1$ [31]. Since $(Au)_i = \frac{1}{h^2}\Big(- u_{i+1} + 2u_i - u_{i-1}\Big)$, we can find the eigenvalues using simple trigonometric identities, as:

$$\begin{aligned} (AV^{(k)})_i &= \frac{1}{h^2}\left[- \sin\left(\frac{k\pi(i+1)}{n}\right) + 2\sin\left(\frac{k\pi i}{n}\right) - \sin\left(\frac{k\pi(i-1)}{n}\right)\right] \\ &= \frac{2}{h^2}\left[\sin\left(\frac{k\pi i}{n}\right) - \cos\left(\frac{k\pi}{n}\right)\sin\left(\frac{k\pi i}{n}\right)\right] \\ &= \frac{2}{h^2}\left[1 - \cos\left(\frac{k\pi i}{n}\right)\right]\sin\left(\frac{k\pi i}{n}\right) \\ &= \frac{4}{h^2}\sin^2\left(\frac{k\pi}{2n}\right)V_i^{(k)}. \end{aligned}$$

So, $\frac{4}{h^2}\sin^2\left(\frac{k\pi}{2n}\right)$ is the eigenvalue of A associated with the eigenvector $V^{(k)}$. Now, for the weighted Jacobi method applied to this system as in [7], we have

$$v^{(k+1)} = v^{(k)} + \omega D^{-1} r^{(k)}.$$

Using the residual equation, we get

$$e^{(k+1)} = e^{(k)} + \omega D^{-1} A e^{(k)}$$
$$e^{(k+1)} = (I - \omega D^{-1} A) e^{(k)}.$$

Let $R_\omega = (I - \omega D^{-1} A)$ be the iteration matrix for weighted Jacobi. Since D is the diagonal of A, and $D = \frac{2}{h^2} I$, from (2.39), we have $D^{-1} = \frac{h^2}{2} I$. Hence [34],

$$R_\omega = I - \frac{h^2}{2} \omega A.$$

Note: The weighted Jacobi scheme converges if $|\lambda_k(R_\omega)| < 1$ for $1 \leq k \leq n-1$. We can directly compute

$$\begin{aligned} \lambda_k(R_\omega) &= \lambda_k(I - \tfrac{h^2}{2}\omega A) \\ &= 1 - \omega \tfrac{h^2}{2} \lambda_k(A) \\ &= 1 - \omega \tfrac{h^2}{2} \tfrac{4}{h^2} \sin^2\left(\tfrac{k\pi}{2n}\right) \\ &= 1 - 2\omega \sin^2\left(\tfrac{k\pi}{2n}\right). \end{aligned}$$

Hence, it is clear that the choice of $0 < \omega \leq 1$ guarantees the condition for convergence is fulfilled. Because of the form of $V_i^{(k)}$, k is called the wave number. See [7, 34] for graphs of these vectors for various wave numbers on a mesh of n nodes. By simple graphing, we can see the eigenvectors of A are smooth for $1 \leq k \leq \frac{n}{2}$, and we call these low-frequency modes, while they are oscillatory for $\frac{n}{2} \leq k \leq n-1$, and we call these high-frequency modes [7]. To find the optimal choice of ω, we graph $\lambda_k(R_\omega)$ for $\omega = \frac{1}{2}, \frac{2}{3},$ and 1, see Figure 2.2.

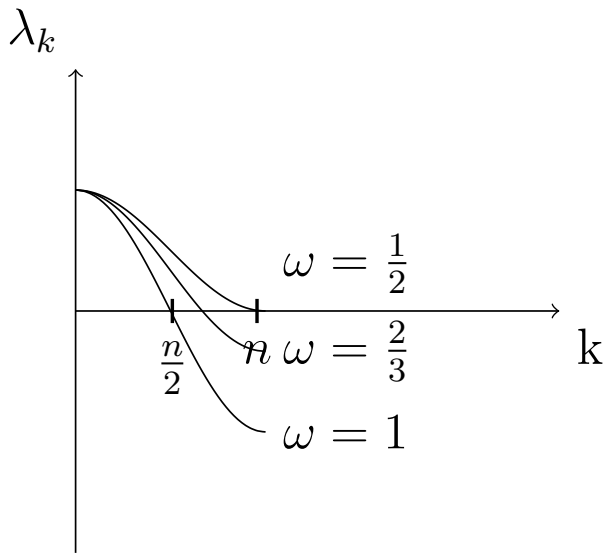


Figure 2.2: Eigenvalues of R_ω

The best possible ω will be that which results in $\lambda_k(R_\omega)$ being as small as possible. In [34], it is found that, for the smoothest modes, when $1 \leq k \leq \frac{n}{2}$, the choice of ω has only a small effect on reducing the error. However, the choice of ω can make a big difference on the error reduction for the oscillatory modes, $\frac{n}{2} \leq k \leq n-1$.

To get the best convergence over the high-frequency modes, we require that

$$\lambda_{\frac{n}{2}}(R_\omega) = -\lambda_n(R_\omega)$$

as in [7] noting that $\lambda_n(R_\omega)$ serves only as an approximation to the true eigenvalue $\lambda_{n-1}(R_\omega)$. Since,

$$\lambda_{\frac{n}{2}}(R_\omega) = 1 - 2\omega \sin^2\left(\frac{\pi}{4}\right) \qquad \text{and} \qquad \lambda_n(R_\omega) = 1 - 2\omega \sin^2\left(\frac{\pi}{2}\right),$$

we have

$$\begin{aligned} 1 - 2\omega \sin^2\left(\tfrac{\pi}{4}\right) &= 2\omega \sin^2\left(\tfrac{\pi}{2}\right) - 1 \\ 1 - \omega &= 2\omega - 1 \\ \omega &= \tfrac{2}{3}. \end{aligned}$$

So, from the perspective of the high-frequency modes, the best choice of ω for

weighted Jacobi is $\frac{2}{3}$. For this ω,

$$\begin{aligned}\lambda_k(R_\omega) &= 1 - 2\left(\tfrac{2}{3}\right)\sin^2\left(\tfrac{k\pi}{2n}\right) \\ &= 1 - \tfrac{4}{3}\sin^2\left(\tfrac{k\pi}{2n}\right).\end{aligned}$$

Using the fact that $\frac{1}{2} \leq \sin^2\left(\frac{k\pi}{2n}\right) \leq 1$ for $\frac{n}{2} \leq k \leq n-1$, we have

$$\begin{aligned}&\lambda_n(R_{\frac{2}{3}}) \leq \lambda_k(R_{\frac{2}{3}}) \leq \lambda_{\frac{n}{2}}(R_{\frac{2}{3}}) \\ &1 - \tfrac{4}{6} \leq 1 - \tfrac{4}{3}\sin^2\left(\tfrac{k\pi}{2n}\right) \leq 1 - \tfrac{4}{3} \\ &\tfrac{-1}{3} \leq 1 - \tfrac{4}{3}\sin^2\left(\tfrac{k\pi}{2n}\right) \leq \tfrac{1}{3}\end{aligned}$$

Hence, $|\lambda_k| \leq \frac{1}{3}$ for $\frac{n}{2} \leq k \leq n-1$. This is often called the smoothing factor [7], and it indicates that with each sweep (iteration) of the relaxation method, the error is reduced by a factor of at least 3 over the oscillatory modes. After a certain number of sweeps, the oscillatory error can always be effectively eliminated by this smoothing effect. Thus, we seek to implement a method that reduces the error in the smooth modes, since relaxation does not greatly affect these modes [34]. From this motivation, multigrid algorithms are developed.

Merely from the name of multigrid methods, one can expect to use multiple grids (meshes) in the algorithm. We start with the uniform one-dimensional grid Ω^h, indicating that the mesh size used is h. A coarser mesh to Ω^h is Ω^{2h}, a grid with mesh size $2h$. We continue with this concept to reach a mesh that cannot be coarsened any further. That grid is called the coarsest grid. In Figure 2.3, we illustrate these grids with $n = 12$ and a wave number $k = 4$ as in [7].

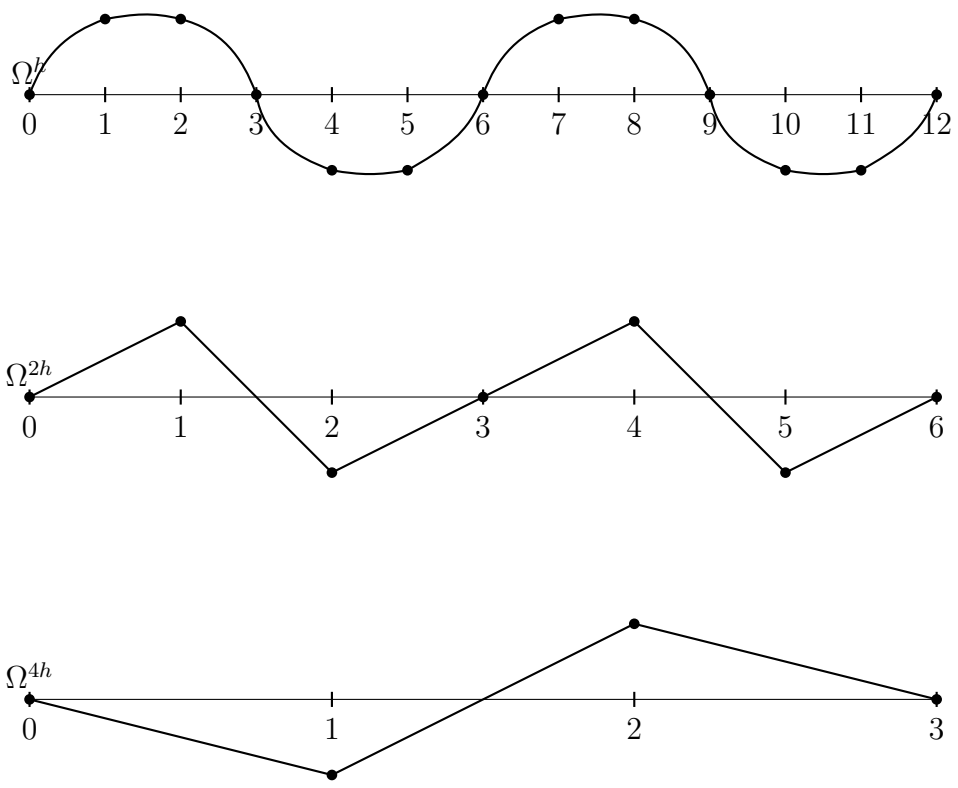


Figure 2.3: Mode with wave number 4 appearing smooth on 12 node mesh and oscillatory on coarser meshes of 6 and 3 nodes, respectively.

Note that the number of nodes is reduced by half as we travel from a fine grid to the next coarser grid. Also, the nodes on any coarse grid are positioned at the even numbered nodes of the finer grid before it. Thus, the components of the k^{th} eigenvector on Ω^h at an even node become

$$V_{2i}^{(k)} = \sin\Big(\frac{(2i)\pi k}{n}\Big) = \sin\Big(\frac{i\pi k}{n/\ 2}\Big),$$

which is exactly the eigenvector of the discretization matrix on Ω^{2h} at the i^{th} node of Ω^{2h} [31]. The same process applies on coarser grids.

It is clear from this example that, as we move further down the grids, the smooth modes on the finer grids are much more oscillatory relative to the coarser grids. This is true only for the smooth modes. We can perform relaxation sweeps on a certain grid until the oscillatory error is reduced as much as possible. When further error

reduction on this grid is not advantageous, we transfer our components to a coarser grid, yielding a more oscillatory error there. This, then, allows the possibility of further error to be reduced with relaxation. This process is repeated to eliminate as much error as we can over the smooth modes using multiple grids. To do this, we need to define how these grids communicate with each other to transfer components. These communication tools are called the restriction and interpolation operators.

Interpolation methods are used to transfer information from a coarse to a finer grid. There are many types of interpolation methods, the easiest of which is called linear interpolation. We will denote the interpolation operator as I_{2h}^{h}, denoting the direction of movement between grids from Ω^{2h} to Ω^{h}. Linear interpolation is defined by taking

$$I_{2h}^{h} v^{2h} = v^{h},$$

where v^{2h} and v^{h} are vectors on Ω^{2h} and Ω^{h}, respectively. For linear interpolation, the operator assigns the values from v^{2h} to the even-numbered nodes on Ω^{h}. For the odd-numbered fine-grid nodes, the average value of the of the two immediate coarse-grid neighbours is used.

For the example above with $h = \frac{1}{12}$, linear interpolation is given by

$$\begin{aligned}
v_0^h &= v_0^{2h} \\
v_1^h &= \tfrac{1}{2}(v_0^{2h} + v_1^{2h}) \\
v_2^h &= v_2^{2h} \\
v_3^h &= \tfrac{1}{2}(v_1^{2h} + v_2^{2h}) \\
\vdots &= \vdots \\
v_{12}^h &= v_6^{2h},
\end{aligned}$$

see Figure 2.4.

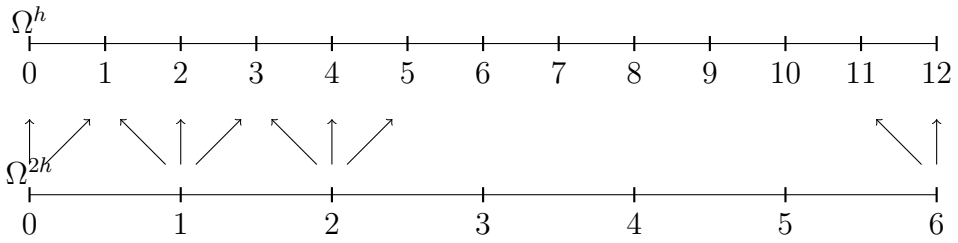


Figure 2.4: Linear interpolation from Ω^{2h} to Ω^h.

In matrix form, we have

$$\frac{1}{2}\begin{bmatrix} 1 & & & & & \\ 1 & 1 & & & & \\ & 2 & & & & \\ & 1 & 1 & & & \\ & & 2 & & & \\ & & \ddots & \ddots & & \\ & & & & 2 & \\ & & & & 1 & 1 \\ & & & & & 1 \end{bmatrix}_{12\times 6} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_6 \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ \vdots \\ v_{12} \end{bmatrix}.$$

Interpolation is not limited to one-dimensional problems, and can be applied to higher-dimensional grids. For an explanation of two-dimensional bilinear interpolation, see [7].

Restriction is the process "opposite" to interpolation, since it transfers values of vectors from a fine grid to a coarser grid. As with interpolation, there are many methods of restriction to choose from. The simplest is called injection. Let I_h^{2h} denote our restriction operator from Ω^{2h} to Ω^h, which is applied as

$$I_h^{2h} v^h = v^{2h}.$$

For injection, this simple matrix merely assigns the values of a fine-grid vector at the even-numbered fine-grid nodes to the corresponding coarse-grid nodes,

$$v_i^{2h} = v_{2i}^h.$$

For many 1D problems , the restriction method used is the full weighting method in place of injection. Full weighting is given by

$$v_i^{2h} = \frac{1}{4}(v_{2i-1}^h + 2v_{2i}^h + v_{2i+1}^h),$$

for $1 \leq i \leq \frac{n}{2} - 1$ [7]. In the following, we give both grid illustrations and matrix forms of both injection and full weighting.

For injection, the operation can be visualized on the grid as in Figure 2.5, with matrix form

$$\frac{1}{2}\begin{bmatrix} 1 & & & & & & & & & & & \\ 0 & 0 & 1 & & & & & & & & & \\ 0 & 0 & 0 & 0 & 1 & & & & & & & \\ & & & & & \ddots & & & & & & \\ & & & & & & & & & & & 1 \end{bmatrix}_{6\times 12} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ \vdots \\ v_{12} \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_6 \end{bmatrix}.$$

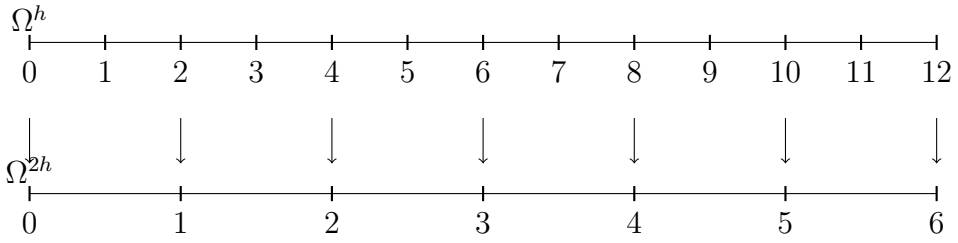


Figure 2.5: Injection from Ω^h to Ω^{2h}.

For full weighting, the operation can be visualized on the grid as in Figure 2.6.

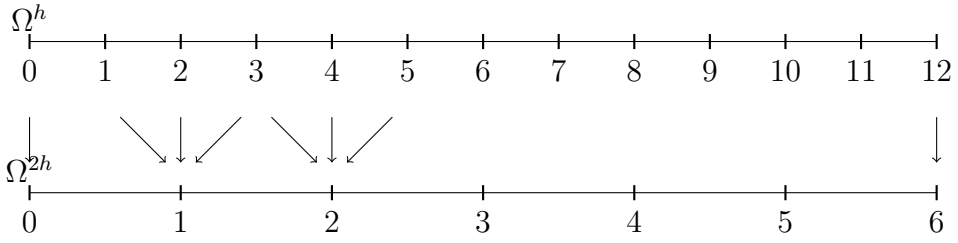


Figure 2.6: Full weighting restriction from Ω^h to Ω^{2h}.

In matrix form, it becomes

$$\frac{1}{4}\begin{bmatrix} 1 & & & & & & & & & & & \\ & 1 & 2 & 1 & & & & & & & & \\ & & & 1 & 2 & & & & & & & \\ & & & & & \ddots & & & & & & \\ & & & & & & & 2 & 1 & & & \\ & & & & & & & & & 1 & & \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ \vdots \\ v_{12} \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_6 \end{bmatrix}.$$

An interesting relation comes from the use of full weighting restriction, which is

$$I_{2h}^{h} = c(I_{h}^{2h})^{T},$$

where c is a real constant. This is called a variational property and will be revisited later.

Now, we are ready to tie our pieces together to present the two-grid algorithm. We start off by using a relaxation method and an initial guess, v^h, to perform m_1 sweeps on the finest grid, Ω^h, to reduce as much oscillatory error as possible, until the error can not be effectively reduced any further. Then, we compute the residual of our problem on this grid, $r^h = f - A^h v^h$, which we restrict to Ω^{2h} using a restriction operator, I_h^{2h}, by computing $I_h^{2h} r^h = r^{2h}$. We next find the exact error, e^{2h}, on Ω^{2h}, by solving the residual equation, $A^{2h} e^{2h} = r^{2h}$. The vector that is interpolated then is this error approximation using the operator, I_{2h}^h, by computing $I_{2h}^h e^{2h} = e^h$. We use this vector to update our approximation, v^h, calculated earlier, as $v^h \leftarrow v^h + e^h$. Finally, we again perform m_2 relaxation sweeps on our problem with the initial guess now being the corrected approximation vector. Here, m_1 and m_2 are usually 1,2, or 3 [7]. They might be equal or different, depending on the problem at hand.

Up to now, we have not properly defined the matrix A^{2h} on Ω^{2h}. Using the interpolation step in the two-grid algorithm, we aim to approximate the error on Ω^h as $e^h = I_{2h}^h e^{2h}$. Multiplying this equation by A^h on both sides, we get a version of the residual equation,

$$r^h = A^h e^h = A^h I_{2h}^h e^{2h}.$$

Now, we can restrict this by applying I_h^{2h} on both sides to get

$$I_h^{2h} r^h = I_h^{2h} A^h I_{2h}^h e^{2h},$$

where we define $A^{2h} = I_h^{2h} A^h I_{2h}^h$. This definition is referred to as the Galerkin condition [31] when $I_h^{2h} = c(I_{2h}^h)^T$.

If e_i^{2h} is the i^{th} elementary vector defined on Ω^{2h}, we can calculate A^{2h} columnwise as

$$e_i^{2h} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}_{(\frac{n}{2}\times 1)} \rightarrow I_{2h}^h e_i^{2h} = \begin{bmatrix} 0 \\ \frac{1}{2} \\ 1 \\ \frac{1}{2} \\ 0 \end{bmatrix}_{(n\times 1)} \rightarrow A^h I_{2h}^h e_i^{2h} = \begin{bmatrix} \\ \frac{-1}{2h^2} \\ 0 \\ \frac{1}{h^2} \\ 0 \\ \frac{-1}{2h^2} \\ \end{bmatrix}_{(n\times 1)} \rightarrow I_h^{2h} A^h I_{2h}^h e_i^{2h} = \begin{bmatrix} \\ \frac{-1}{4h^2} \\ \frac{1}{2h^2} \\ \frac{-1}{4h^2} \\ \end{bmatrix}_{(\frac{n}{2}\times 1)}.$$

Why is this a plausible definition of A^{2h}? From this calculation, we see that A^{2h} has a stencil of $\frac{1}{4h^2}[-1\ 2\ -1]$, while we know from discretization of the model problem, the stencil of A^h is $\frac{1}{h^2}[-1\ 2\ -1]$. That these coincide suggests that our definition of $A^{2h} = I_h^{2h} A^h I_{2h}^h$ is reasonable.

The problem with two-grid correction schemes is that we are still faced with a complex coarse grid problem on Ω^{2h} if our original problem is large, which usually it is [44]. To address this, we apply the two-grid approach recursively leading to multigrid methods. We follow the notation in [7], where the restricted residual is denoted as f^{2h} not r^{2h} and the error is denoted as u^{2h} instead of e^{2h}. The first multilevel scheme is called the V-cycle scheme, recursively defined as:

Multigrid V- cycle method recursion:
$V^h(v^h, f^h) \rightarrow v^h$

1-Relax m_1 times on $A^h u^h = f^h$ using v^h as an initial guess, calling result u^h.
2-If Ω^h is the coarsest level , go to step 4.
else: (a) compute the residual, $r^h = f^h - A^{2h}u^h$ and restrict $f^{2h} = I_h^{2h} r^h$.
(b) set $v^{2h} = 0^{2h}$.
3-Correct the approximation by computing $V^{2h}(v^{2h}, f^{2h}) \rightarrow v^{2h}$.
4-Relax m_2 times on $A^h u^h = f^h$ with the initial guess $u^h + I_{2h}^h v^{2h}$.

This is called the V-cycle because of the pattern of travelling all the way to the coarsest grid then back up to the finest, resulting in a V motion as shown in Figure 2.7. Other methods have the same details with different patterns. For example, the W-cycle results from travelling to the coarsest grid and then up one level, then down one level, then up 2 levels and so on, resulting in a W pattern. Full multigrid (FMG) is another type of scheme, which begins at the coarsest grid. We omit the details of this cycle here. We will refer later on to the V-cycle with $m_1$and m_2 relaxation sweeps as $V(m_1, m_2)$.

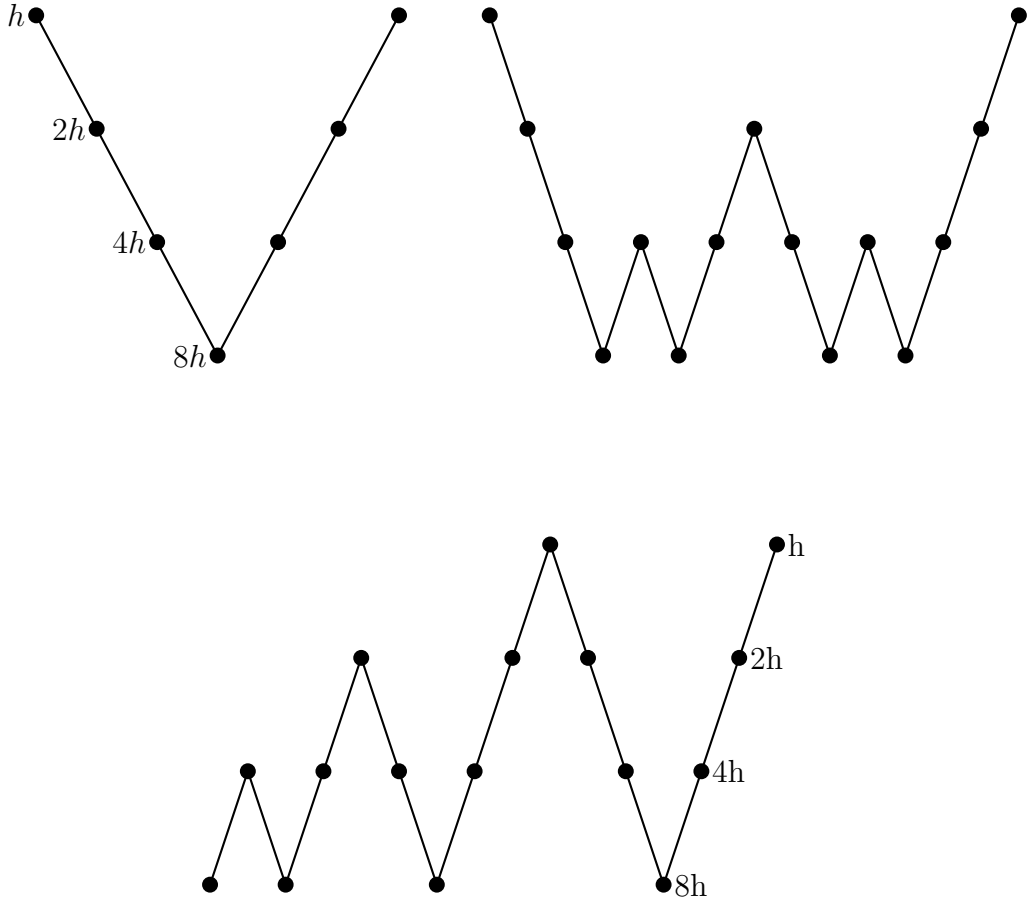


Figure 2.7: Multigrid schemes:(a) upper left corner: V-cycle, (b) upper right corner: W-cycle, (c) below: FMG cycle.

Cost of multigrid methods:

Storage cost:

For the V-cycle, each grid in the scheme stores 2 vectors: the current approximation of the solution (or error on coarse grids) and the right-hand side vector associated with that grid. If we consider a d dimensional grid with n nodes in each dimension, we would have n^d total number of nodes on Ω^h. Hence, on Ω^h, we need $2n^d$ storage locations for both vectors. From the fact that each time we coarsen we reduce the amount of nodes by half in each dimension, on Ω^{2h} we will need $\frac{1}{2^d}$(storage of Ω^h), on Ω^{4h} we need $\frac{1}{4^d}$(storage of Ω^h) and so forth. So, summing the geometric series, we get

$$\begin{aligned} \text{V-cycle storage} &= 2n^d(1+\tfrac{1}{2^d}+\tfrac{1}{4^d}+\cdots+\tfrac{1}{L^d}) \\ &= 2n^d(1+\tfrac{1}{2^d}+\left(\tfrac{1}{2^d}\right)^2+\cdots+\left(\tfrac{1}{2^d}\right)^L) \\ &< \tfrac{2n^d}{1-2^{-d}}, \end{aligned}$$

Note:
1-For $d = 1$, the storage needed on $\Omega^h = 2n$, and the total cycle storage cost is bounded by $\frac{2n}{1-\frac{1}{2}} = 4n$.
2-For $d = 2$, the storage needed on $\Omega^h = 2n^2$, and the total cycle storage cost is bounded by $\frac{2n^2}{1-\frac{1}{4}} = \frac{8}{3}n^2$.
3-For $d = 3$, the storage needed on $\Omega^h = 2n^3$, and the total cycle storage cost is bounded by $\frac{2n^3}{1-\frac{1}{8}} = \frac{16}{7}n^3$.

Note that the storage costs of MG decrease relative to the fine-grid storage cost for higher dimensions, d.

Computational cost:
We define a work unit (**wu**) as the cost of performing 1 relaxation sweep on Ω^h. In our cost estimates, interpolation and restriction costs are disgarded, since they typically account for less than 20% of the total computational cost of the cycle [5]. For the V-cycle, relaxation is performed on each grid $m = m_1 + m_2$ times. Again, summing the geometric series, we have

$$\begin{aligned} \text{V-cycle computation cost} &= m(1+\tfrac{1}{2^d}+\tfrac{1}{4^d}+\cdots+\tfrac{1}{(2^L)^d}) \\ &= m(1+\tfrac{1}{2^d}+\left(\tfrac{1}{2^d}\right)^2+\cdots+\left(\tfrac{1}{2^d}\right)^L) \\ &< \tfrac{m}{1-2^{-d}}\ \textbf{wu}. \end{aligned}$$

1-For V(1,1) with $d = 1$, $m = 2$, the total cycle computation cost is bounded by $\frac{2}{1-\frac{1}{2}} = 4$ **wu**.
2-For V(1,1) with $d = 2$, $m = 2$, the total cycle computation cost is bounded by $\frac{2}{1-\frac{1}{4}} = \frac{8}{3}$ **wu**.
3-For V(2,1) with $d = 2$, $m = 3$, the total cycle computation cost is bounded by $\frac{3}{1-\frac{1}{4}} = 4$ **wu**.

Convergence of MG methods:
Numerical experiments with MG show the high efficiency of the method, but it is a

difficult process to prove convergence of MG theoretically. Two-grid convergence is easier to show and can often be generalized to multigrid convergence.

Now, to establish convergence, we define some notation. Our model problem, $-u'' = f$, is called the continuous problem and $A^h u^h = f^h$ is the discrete problem on Ω^h. We define the global error as $E^h = u(x_i) - u_i^h$, for $1 \leq i \leq n-1$ where $u(x_i)$ is the exact solution of the continuous problem and u_i^h is the exact solution to the discrete problem. Define the algebraic error as $e^h = u^h - v^h$, where v^h is the approximate solution to the discrete problem.

The L_2 norm of the global error is typically bounded by $||E^h|| \leq Kh^p$, where K and p are a positive constant and positive integer, respectively, depending on the discretization [7]. This error is hard to calculate exactly since we do not know the exact solutions of the problem, so the algebraic error is somehow more tractable. To show convergence, we look to find a bound for $u - v^h$, where $u = (u(x_1), u(x_2), \ldots, u(x_{n-1}))^T$, i.e. , we aim to satisfy

$$||u - v^h|| < \epsilon, \tag{2.41}$$

for some given ϵ.

By the triangle inequality, we have

$$||u - v^h|| \leq ||u - u^h|| + ||u^h - v^h|| = ||E^h|| + ||e^h||.$$

Thus, condition (2.41) is true if $||E^h|| + ||e^h|| < \epsilon$. This is possible if we choose $||E^h|| < \frac{\epsilon}{2}$ and $||e^h|| < \frac{\epsilon}{2}$. For $||E^h|| < \frac{\epsilon}{2}$, using the standard bound on global error, we have

$$Kh^p < \frac{\epsilon}{2} \longrightarrow h < \left(\frac{\epsilon}{2K}\right)^{\frac{1}{p}}.$$

Let $h^* = \left(\frac{\epsilon}{2K}\right)^{\frac{1}{p}}$, and choose $h < h^*$. Thus, we can use a desired global error tolerance to determine the grid spacing [31]. To reduce the algebraic error to a similar level determines the number of iterations that the relaxation method should do on each grid and the number of MG cycles in total.

As mentioned earlier on a grid with n elements, relaxation damps oscillatory modes of error, for modes $\frac{n}{2} \leq k \leq n-1$. Since this is true on each grid in the MG hierarchy, using relaxation on coarse grids damps the oscillatory modes on that grid

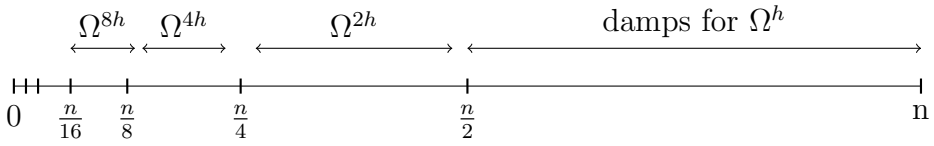


Figure 2.8: damping process on grids.

Recall, the smoothing factor is the largest magnitude eigenvalue of the iteration matrix over the oscillatory modes. Above, we saw that the smoothing factor for relaxation is small and independent of the grid size h. Since all modes are oscillatory on some grid, the overall convergence rate for MG is expected to be small and independent of h. Figure 2.8 shows this heuristic argument for multigrid convergence.

Before moving forward, we note how eigenvectors change with the value of k. Recall, the $2i^{th}$ component of the k^{th} eigenvector of A is $\sin\left(\frac{2\pi ki}{n}\right)$ for wave numbers $1 \leq k < \frac{n}{2}$ on Ω^h. We notice this k^{th} mode is also the k^{th} mode on Ω^{2h}, as

$$\sin\left(\frac{2\pi ki}{n}\right) = \sin\left(\frac{\pi ki}{n/2}\right).$$

For $k = \frac{n}{2}$, this mode on Ω^h is the zero vector on Ω^{2h} because

$$\sin\left(\frac{2\pi(\frac{n}{2})i}{n}\right) = \sin\left(\pi i\right) = 0.$$

The eigenvectors for $\frac{n}{2} < k \leq n$ undergo a process called aliasing on the coarse grid, meaning that these modes on Ω^h appear as modes with wave numbers $n-k$ on Ω^{2h}. Let $k = n - \hat{k}$, where $1 \leq \hat{k} < \frac{n}{2}$. Then,

$$\begin{aligned}
\sin\left(\tfrac{2\pi i(n-\hat{k})}{n}\right) &= \sin\left(2\pi i - \tfrac{2\pi i\hat{k}}{n}\right) \\
&= \sin(2\pi i)\cos\left(\tfrac{2\pi i\hat{k}}{n}\right) - \cos(2\pi i)\sin\left(\tfrac{2\pi i\hat{k}}{n}\right) \\
&= -\sin\left(\tfrac{2\pi i\hat{k}}{n}\right).
\end{aligned}$$

So, high-frequency modes on Ω^h appear as low-frequency modes on Ω^{2h}. The same is true on coarser grids.

To establish convergence, we need to consider how restriction and interpolation map modes between grids. Restricting a mode with $1 \leq k < \frac{n}{2}$ using full weighting we get the i^{th} component

$$\begin{aligned}(I_h^{2h}V^{(k)})_i &= \tfrac{1}{4}(V_{i-1}^{(k)} + 2V_i^{(k)} + V_{i+1}^{(k)}) \\ &= \tfrac{1}{2}\sin\left(\tfrac{2k\pi i}{n}\right) + \tfrac{1}{4}\Big[\sin\left(\tfrac{2k\pi(i-1)}{n}\right) + \sin\left(\tfrac{2k\pi(i+1)}{n}\right)\Big] \\ &= \tfrac{1}{2}\sin\left(\tfrac{2\pi ik}{n}\right)\Big(1 + \cos\left(\tfrac{k\pi}{n}\right)\Big).\end{aligned}$$

So, if we denote $\hat{V}^{(k)}$ as the k^{th} mode on Ω^{2h}, we have $I_h^{2h}V^{(k)} = \mu_k(I_h^{2h})\hat{V}^{(k)}$, where $\mu_k(I_h^{2h}) = \frac{1}{2}\Big(1 + \cos\left(\frac{k\pi}{n}\right)\Big)$. Applying the same process to modes with $\frac{n}{2} < k < n$, we find that $I_h^{2h}V^{(k)} = \mu_{\hat{k}}(I_h^{2h})\hat{V}^{(\hat{k})}$, where $\mu_{\hat{k}}(I_h^{2h}) = \frac{-1}{2}\Big(1 + \cos\left(\frac{k\pi}{n}\right)\Big)$, where $\hat{V}^{(\hat{k})}$ is the $\hat{k}^{th}$ mode on Ω^{2h}.

Considering the effect of restriction on modes with $1 \leq k < \frac{n}{2}$, we can write this as $I_h^{2h}(c_kV^{(k)} + c_{\hat{k}}V^{(n-k)}) = (c_k\mu_k + c_{\hat{k}}\mu_{\hat{k}})\hat{V}^{(k)}$, where c_k and $c_{\hat{k}}$ are constants associated with k^{th} and $\hat{k}^{th}$ modes, respectively. Using similar analysis, we can consider interpolation of the k^{th} mode on Ω^{2h} using linear interpolation to get $I_{2h}^h\hat{V}^{(k)} = c_kV^{(k)} + s_kV^{(n-k)}$, where $c_k > s_k$ for $1 \leq k < \frac{n}{2}$. The first term, $c_kV^{(k)}$, is a smooth component of the solution while the second term is an oscillatory component [31].

Thus, for convergence, we must analyse actions of relaxation and coarse-grid correction on a linear combination of both $V^{(k)}$ and $V^{(n-k)}$. Doing this, the two-grid cycle with $m = 2$ total sweeps of relaxation using weighted Jacobi with $\omega = \frac{2}{3}$ gives an error reduction of a factor of about 0.1 per iteration [31]. In general, since we know that V-cycles visit each grid two times per cycle, the cost of a d-dimensional V-cycle with n^d nodes is $O(n^d)$. Assuming we had a V-cycle with convergence rate γ, then for the V-cycle to get from an initial, $O(1)$ error, to the order of the global error, $O(n^{-p})$, the total number of cycles needed must be equal to $O(\log n)$. Thus, total multigrid cost for V-cycles is $O(n^d \log n)$. FMG costs a little more per cycle than V-cycles but gives a total cost bound without the logarithmic factor. For more detail on this, see [7].

Algebraic Multigrid:

In the previous subsection, we presented geometric multigrid methods. These methods use structured grids on which the method is implemented. A natural question to ask is what if we are faced with a problem where no structured grids exist. Does multigrid fail then? The answer is no, multigrid, in general, does not fail, only geometric multigrid (GMG) is no longer suitable. In the early 80's, this was a topic of concern, and algebraic multigrid methods arose [7, 44]. Algebraic multigrid (AMG) methods are multigrid methods that depend only on the entries of the underlying matrices, A, of the problem, with no requirement of geometric grids. In the following discussion, we assume our problem is of the form $Au = f$. We also assume A to be a symmetric, positive-definite matrix.

Recall that our main goal is to find the best approximation possible for our problem, i.e., reducing the error as much as possible using smoothing and coarse-grid correction. Smoothing reduces oscillatory error but not smooth error, and the multigrid idea is used to reduce these smooth errors. So, we must find a way of smoothing and performing coarse-grid correction on non-geometric "grids". One main difference between GMG and AMG is that in GMG, the coarsening method is fixed by predefining a certain number of coarser grids, and we are left to choose the appropriate smoothing method. However, in AMG, it is the opposite, where a relaxation method is chosen before-hand and fixed throughout the process, while we are left to choose the best coarse-grid correction method, by choosing appropriate coarse levels and interpolation operators. Often in AMG, the relaxation method used is GS.

We note that AMG clearly has a wider range of possible application than GMG, since it may be applied to structured and unstructured grids. Also, unlike GMG, there are two phases of AMG: a setup phase and a solution phase. Setup phases are done at the begining of the computation, where the problem is analysed, all coarse levels are produced and operators are all defined. Much of the cost of AMG comes from this phase, but it is often worth the price [35]. The solution phase is simple, and usually follows the same cycling as in GMG.

We note that the reader should keep in mind that, in the following discussion of

AMG, we use the notations H and h to only indicate the level we are referring to. This does not have any connection to the grid spacing, since geometric grids are not defined in AMG. We will define coarsening, transfer operators and coarse-grid operators. To specify the setup phase, we start by defining a coarsening method. Since no physical grids exist in AMG, instead of grids, we have levels, and instead of grid nodes, we have variables or points. We use Ω^h to denote the fine level with a set of points $\{1, 2, 3 \ldots, n\}$. To coarsen, we split Ω^h as $\Omega^h = C^h \cup F^h$, where the points in C^h are called C-variables and make up the coarse level. Points in F^h are called F-variables and lie only on the fine level, being the complement of the C-variables on the fine level. We define our coarser level as $\Omega^H = C^h$.

We now need to find a way of choosing these C-variables. The number of C-variables chosen should be as small as possible in order to keep the cost of AMG low, since costs for computing A^H and subsequent calculations depends on the number of C-variables. Also, keeping in mind that the coarse level must be able to represent the error after relaxation restricted from the fine level, the coarse set should not be too small. In AMG, the coarsening process is usually carried out in as uniform a way as possible, with C-variables on the fine level surrounded by F-variables. This distribution has been found to support a better interpolation operator resulting in faster convergence [35].

There are many methods of corsening. Here, we will present the standard Ruge-Stüben coarsening algorithm. In this, and other coarsening methods, connections between variables play a big role. We say that a point $i \in \Omega^h$ is connected directly (coupled) to another point $j \in \Omega^h$ if $a^h_{ij} \neq 0$, where a^h_{ij} are elements of the matrix A^h. These sets of points are referred to as the neighbourhood of node i denoted as $N^h_i = \{j \in \Omega^h : j \neq i, a^h_{ij} \neq 0\}$, for $i \in \Omega^h$. These connections could be strong or weak, negative or positive. For the M-matrices typical of discretizations of diffussion equations, most strong connections are negative. A variable i is said to be strongly connected (strongly coupled) to another point, j, if

$$-a_{ij} \geq \epsilon \max_{k \text{ s.t. } a_{ik}<0} |a_{ik}|,$$

where $0 < \epsilon < 1$ is a fixed constant usually taken to be equal to 0.25 [35]. Any other

connections are called weak connections. Let $S_i = \{j \in N_i : i \text{ is strongly coupled to } j\}$ be the set of all strongly coupled variables to i. This set defines the matrix of strong connections, S, with elements

$$S_{ij} = \begin{cases} 1, & \text{if } i \text{ is strongly connected to } j \\ 0, & \text{otherwise .} \end{cases}$$

Since the set S_i need not be symmetric, even if A is, we also consider S_i^T, which is the set of strong transpose couplings of i, containing all points j that are strongly coupled to i,

$$S_i^T = \{j \in \Omega : i \in S_j\}.$$

The steps of coarsening are done in 2 stages or passes. The first pass starts by defining one i point to be a C-variable. Then, all the strongly coupled points to that first C-variable are assigned to be F-variables. We repeat this process, picking another C-variable from the set of unassigned points and marking its unassigned neighbours as F-variables until all points are assigned to either F or C-variables. But, in what order do we pick the C-variables? For this, we use a measure of importance,

$$\lambda_i = \text{number of elements in } (S_i^T \cap U) + \text{ number of elements in } (S_i^T \cap F),$$

for each $i \in U$, the set of currently unassigned points. This measure is calculated and updated for each point in the fine level in the coarsening phase. The coarsening algorithm becomes:

1. Start by having the sets C and F both empty and U be the set of all points, $\{1, 2, \ldots, n\}$.
2. Find λ_i for all points in U.
3. Find the unassigned point, $i \in U$, which has the largest λ_i.
4. Add i to the set of coarse points, C.
5. Find all j points that are strongly connected to this i and add them to the set of fine points.

6. For each of these j's, find every point k strongly connected to j and increase λ_k by 1.

7. If U is empty, then the partitioning is done. If not, repeat from step 3.

Often, a "second pass" is used to improve the equality of the coarsening. The second pass involves going back to the points assigned in the first pass and checking the connections between F-points. If 2 F-points have a strong connection between them, but no common strongly connected C-point, we usually add one of the F-points to the coarse set. This ensures that it is possible to construct a reasonable interpolation operator in the next stage of the setup.

Remarks:
1- In the first pass, C-points are assigned in a way that there are no direct connections between them but, in the second pass, such direct connections might arise.
2- This standard coarsening method is only effective if any positive connections in the matrix are small enough to be neglected. If they are not, another method of coarsening should be chosen.

For multilevel coarsening, this method is applied to the coarse level, Ω^H, to define an even coarser level, Ω^{H_2}, and so on until, like GMG, we reach a level where we cannot coarsen further and the discrete operator is small enough that a direct solve can be used.

With each coarse level, we need to define an interpolation operator. Like coarsening, there are many ways of defining an interpolation operator. We present direct interpolation, which assumes the standard coarsening algorithm is used and use I_H^h to denote interpolation in AMG from a coarse level to the next finer level. For interpolation, we have to transfer an error vector on the coarse level to an error vector on the fine level. If a C-variable, j, has a strong coupling with $i \in F$, then the coarse-grid value at point j strongly affects the value of e_i on the fine level [7].

To determine interpolation, we consider the condition, $||S^h e||_A \approx ||e||_A$, where $||.||_A$ is the A-norm, $||u||_A^2 = (u,u)_A = (Au,u)$ and S^h is the error-propagation operator

for the relation scheme used in AMG. For Jacobi smoothing, for example, we write $||(I - wD^{-1}A)e||_A \approx ||e||_A$. This is equivalent to

$$(D^{-1}Ae, Ae) \ll (e, Ae),$$

where $\ll$ denotes much less than. Along with the fact that A is symmetric and using $Ae = r$, we then have

$$(D^{-1}r, r) \ll (e, Ae),$$

or $||r||_{D^{-1}} \ll ||e||_A$. Now, because $||e||_A^2 = (Ae, e) = (D^{-\frac{1}{2}}Ae, D^{\frac{1}{2}}e) \leq ||r||_{D^{-1}}||e||_D$, if $||S^h e||_A \approx ||e||_A$, $||e||_A \ll ||e||_D$. Writing $(Ae, e) = \frac{1}{2}\sum_{i,j}(-a_{ij})(e_i - e_j)^2 + \sum_{i,j} a_{ij}e_i^2$, we reach the condition

$$\sum_i \sum_{i \neq j} |a_{ij}|(e_i - e_j)^2 \ll \sum_i a_{ii}e_i^2$$

by assuming $\sum_j a_{ij} \approx 0$ as is typical for discretized diffusion equations. Eliminating the summation over i and dividing both sides gives

$$\sum_{j \neq i} \left(\frac{|a_{ij}|}{a_{ii}}\right) \left(\frac{e_i - e_j}{e_i}\right)^2 \ll 1, \tag{2.42}$$

for $1 \leq i \leq n$ [7]. Note that, $\frac{|a_{ij}|}{a_{ii}}$ can be large or small. If it is large, then the second term, $\left(\frac{e_i - e_j}{e_i}\right)^2$ must be small, i.e $e_i \approx e_j$. This motivates consideration of error that varies slowly in the direction of strong connections.

Following [7, 35], for point $i \in F$, we can divide the points in the neighbourhood set, $j \in N_i$, to 3 different categories.

1. j is a coarse-level point with a strong connection to i. This set is denoted as P_i and called the interpolatary set for i.

2. j is an F-variable and has a strong connection to i. This set is denoted by D_i^s.

3. j is either in F or a C-variable and has weak connections to i. This set is denoted as D_i^w.

This division of N_i is key to the definition of interpolation. Direct interpolation is defined to take form

$$(I_H^h e^H)_i = \begin{cases} e_i^h, \text{ if } i \in C \\ \sum_{j \in P_i} w_{ij} e_j^h, \text{ if } i \in F, \end{cases}$$

where w_{ij} are called the interpolation weights. To find w_{ij}, we notice that when $Ae \approx 0$, we have the componentwise relation

$$a_{ii} e_i \approx - \sum_{j \in N_i} a_{ij} e_j.$$

If we were to divide the summation in this equation into the 3 kinds of points in N_i, we would have

$$a_{ii} e_i \approx - \sum_{j \in P_i} a_{ij} e_j - \sum_{j \in D_i^s} a_{ij} e_j - \sum_{j \in D_i^w} a_{ij} e_j.$$

So, to define w_{ij}, we should try to only have a summation over only P_i on the right-hand side above. For $j \in D_i^w$, since connections are weak to i, we can estimate e_j by e_i. This is not the best approximation, but we can do this because the effect of this summation on i is not significant. We can, thus, move this summation to the left-hand side.

For $j \in D_i^s$, replacing e_j is more tricky. One thing we can do is to use the following estimate of e_j

$$e_j \approx \frac{\sum_{k \in P_i} a_{jk} e_k}{\sum_{k \in P_i} a_{jk}}.$$

This approximation uses points k in the coarse variable set, C, within a weighted average of those points strongly connected to both i and j [7]. We can then input this

into the above, giving

$$w_{ij} = \frac{-a_{ij} + \sum_{m \in D_i^s} \left(\frac{a_{im} a_{mj}}{\sum_{k \in P_i} a_{mk}} \right)}{a_{ii} + \sum_{n \in D_i^s} a_{in}}.$$

Now, with interpolation defined, we simply use the variational property discussed previously for restriction,

$$I_h^H = (I_H^h)^T.$$

For the coarse level matrix, A^H, we also just use the Galerkin condition to define $A^H = I_h^H A^h I_H^h$.

We now have all of the pieces of necessary to construct a two-level AMG. This method is exactly the same as the two-grid GMG with the transfer operators, A^H, and coarsening defined in the AMG environment. Like GMG, this method can be extended to define a multilevel method, with V-,W-, and full multigrid methods defined in exactly the same way as GMG.

The two-level AMG method is as:

1. (Pre-smoothing) Relax on Ω^h using S^h.
2. Restrict the residual from Ω^h to Ω^H, $f^H = I_h^H(f^h - A^h u^h)$.
3. Solve the coarse-level problem $A^H u^H = f^H$ directly.
4. Interpolate the coarse-grid correction to Ω^h from Ω^H.
5. Correct the current approximation on Ω^h.
6. (Post-smoothing) Relax the problem again on Ω^h using the corrected approximation as initial guess.

A useful way to think of AMG is to reorder the system, $A^h u^h = f^h$, based on the connections between F- and C-variables. If we were to define a permutation matrix,

P, that reorders the degrees of freedom to give the F-points first and C-points second, P^TAP, P^Tu and P^Tf can be written in block form as

$$\begin{bmatrix} A_{FF} & A_{FC} \\ A_{CF} & A_{CC} \end{bmatrix} \begin{bmatrix} u_F \\ u_C \end{bmatrix} = \begin{bmatrix} f_F \\ f_C \end{bmatrix}. \tag{2.43}$$

We can also do the same for the interpolation and restriction matrices,

$$I_H^h = \begin{bmatrix} I_{FC} \\ I_{CC} \end{bmatrix} = \begin{bmatrix} W_{FC} \\ I \end{bmatrix}, I_h^H = \begin{bmatrix} I_{CF} & I_{CC} \end{bmatrix} = \begin{bmatrix} W_{FC}^T & I \end{bmatrix}, \tag{2.44}$$

where I is the identity matrix and W_{FC} comes from the weights of interpolation.

Convergence of AMG:

In the following discussion, we follow [32, 35]. Before we demonstrate convergence, we clarify some notation. Let S^h denote the relaxation operator, such as GS. In the previous subsection, we wrote the error correction step in multigrid as $v \leftarrow v + I_H^h e^h$, where v is the current approximation found by relaxation before coarse-grid correction. Tracing through the coarse-grid correction process, we have an error update of

$$e^h \leftarrow (I - I_H^h (I_h^H A^h I_H^h)^{-1} (I_H^h)^T A^h) e^h,$$

where I is the identity matrix of size same as size of A^h [32]. Since we know $A^H = I_h^H A^h I_H^h$, the coarse-grid correction operator can be written as $T^h = I - I_H^h (A^H)^{-1} (I_H^h)^T A^h$.

We use the following inner products:

$$< u, v >_0 = < Du, v >, \quad < u, v >_1 = < Au, v >, \quad < u, v >_2 = < D^{-1}Au, Av >,$$

where $D = \text{diag}(A)$. The norms $||.||_0$, $||.||_1$ and $||.||_2$ are associated with the above inner products, respectively, such that $||u||_i^2 = < u, u >_i$ for $i = 0, 1, 2$.

Denote $||S^hT^h||_1$ as the two-level convergence factor measured in the A norm.

Theorem 6 *Assume A is SPD, and S^h as defined earlier. Let $||S^he^h||_1^2 \leq ||e^h||_1^2 - \delta||T^he^h||_1^2$, where $\delta > 0$ is independent of h and e^h. Then $\delta \leq 1$ and the convergence factor of a V-cycle, with 1 smoothing sweep after coarse-grid correction is $||S^hT^h||_1 \leq \sqrt{1-\delta}$.*

Theorem 7 *Assume A is SPD, and S^h as defined earlier. Let $||S^he^h||_1^2 \leq ||e^h||_1^2 - \delta||T^hS^he^h||_1^2$, where $\delta > 0$ is independent of h and e^h. Then $\delta \leq 1$ and the convergence factor of a V-cycle, with 1 smoothing sweep before coarse-grid correction , is $||T^hS^h||_1 \leq \frac{1}{\sqrt{1-\delta}}$.*

The proofs of these two theorems are omitted here, but can be found in [35].

We can separate the inequality conditions assumed in both theorems into two different inequalities in many ways. A common separation is to assume

$$\begin{aligned} ||S^he^h||_1^2 &\leq ||e^h||_1^2 - \alpha||e^h||_2^2, \longrightarrow \text{ this is called the smoothing assumption ,} \\ ||T^he^h||_1^2 &\leq \beta||e^h||_2^2, \longrightarrow \text{ this is called the approximation assumption ,} \end{aligned}$$

where α and β are both positive constants and δ in the theorems above is replaced by $\frac{\alpha}{\beta}$ [32, 35]. We note that the smoothing assumption is derived from Theorem 6. If we were to use Theorem 7, we would rewrite this assumption as $||S^he^h||_1^2 \leq ||e^h||_1^2 - \alpha||S^he^h||_2^2$.

For two-level convergence of AMG, using the first form of the smoothing assumption, we can state a convergence theorem as follows:

Theorem 8 *If A and S^h are defined as before, and $\min_{e^H}||e^h - I_H^he^H||_0^2 \leq \beta||e^h||_1^2$, where β is independent of e^h, then $\beta \geq \alpha$ and the two-level convergence factor for AMG is*

$$||S^hT^h||_1 \leq \sqrt{1 - \frac{\alpha}{\beta}}.$$

It is difficult to extend these results from two-level convergence to prove convergence of V-cycle AMG methods. For AMG W- and F- cycles, convergence has been found to be the same as in Theorem 8 if the assumptions hold uniformly on all grids.

So, for V-cycles, we generally need a stronger condition that that used in Theorem 8. Convergence is guaranteed under the condition $||T^h e^h||_1^2 \leq \beta ||e^h||_2^2$ which requires the stronger bound

$$\min_{e^H} ||e^h - I_H^h e^H||_0^2 \leq \beta^2 ||e^h||_2^2.$$

This leads us to the important requirement of defining more accurate interpolation operators. Further details on this are found in [32].

Chapter 3

Methodology

In this work, we analyse the following two-dimensional boundary value problem, with Neumann boundary conditions. We aim to study some iterative methods for solving this problem and compare the costs of these methods. Consider the equation

$$\begin{aligned} -\nabla \cdot (K(x,y)\nabla u(x,y)) &= f(x,y), \qquad && \text{for } (x,y) \text{ in } \Omega \\ (K(x,y)\nabla u) \cdot \overrightarrow{n} &= 0, && \text{for } (x,y) \text{ in } \partial\Omega \end{aligned} \tag{3.1}$$

where Ω is the domain, with boundary $\partial\Omega$ and $\overrightarrow{n}$ is the (outward) unit normal vector on the boundary. Following [10], we set $\Omega = [0,3] \times [0,1]$, and use a source function composed of a sum of Gaussians,

$$Ae^{-\left[\frac{(x-x_0)^2}{2\delta_x^2}+\frac{(y-y_0)^2}{2\delta_y^2}\right]},$$

where $\delta = (\delta_x, \delta_y)$ is the variance. Here, we set $\delta_x = \delta_y = 0.05$ and consider four terms, associated with the four corners of our domain, with associated weights $\{1, 2, 3, -6\}$, giving

$$f(x,y) = e^{-[\frac{x^2+y^2}{0.005}]} + 2e^{-[\frac{(x-3)^2+y^2}{0.005}]} + 3e^{-[\frac{(x-3)^2+(y-1)^2}{0.005}]} - 6e^{-[\frac{x^2+(y-1)^2}{0.005}]}.$$

Note that $\iint_\Omega f(x,y) = 0$ and, consequently, (3.1) has a solution, $u(x,y)$, that is unique up to constant shifts. K is called the permeability field and is specified to have a log-normal prior distribution, $K = \exp(\kappa)$, with $\kappa \sim N(0,\Gamma)$, where the

covariance matrix, Γ, is defined via the covariance kernel

$$Cov(\kappa(s), \kappa(s')) = \sigma_u^2 \exp\left(\frac{-1}{2}\left((s-s')^T \Sigma^{-1}(s-s')\right)^{\frac{1.8}{2}}\right),$$

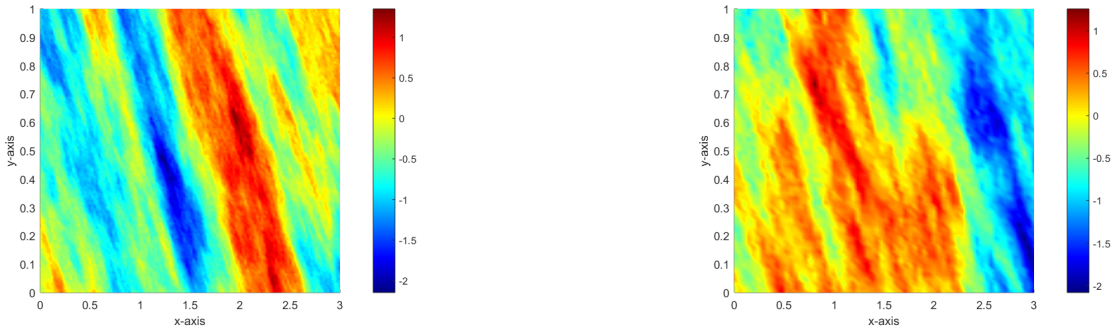
for $s, s' \in \Omega$. In the covariance above, $\sigma_u = 1.15$, and the matrix Σ is given by $\Sigma = \begin{bmatrix} 0.165 & -0.135 \\ -0.135 & 0.165 \end{bmatrix}$.

Denoting the Cholesky factor of covariance defined above by L, the following pseudo-code generates different realizations, K_i,

Algorithm 1 Calculate vectors K_i

for $i = 1$ to total number of samples **do**
  $r_i =$ normal random vector of length n.
  $K_i = \exp(L * r_i)$.
end for

Presented in Figure 3.1 are graphs of $\log(K)$ for a mesh of size 128^2 in (a) and for 64^2 in (b). These are similar to the permeability fields shown in [10].



(a) log(K) for a mesh of 128×128 (b) log(K) for a mesh of 64×64

Figure 3.1: Samples of log(K) of some mesh sizes produced using the algorithm above.

In our research, we use meshes of size n^2, where n is 16, 32, 64 and 128. A limitation arises from the algorithm above. Due to the use of a Cholesky factorization of the dense covariance matrix, it is impractical to generate data for meshes finer than 128×128. We note, however, that this limitation in grid sizes does not affect the conclusions drawn later and can be circumvented using, for example, the stochastic

PDE approach presented in [28]. Going forward, we associate a piecewise linear interpolation of the nodal values of K as a function $K(x, y)$, distinct from the set of vectors of these values generated in this way.

3.1 Finite-element discretization

Before we go further, a very elementary introduction to Finite Element methods is needed. For more detail on this subject see [7, 37]. Finite Element methods are one family of discretization methods. We start by considering the same model problem as (2.31), but with a slight change in boundary conditions,

$$\begin{aligned} -u''(x) &= f(x), \qquad \text{in } \Omega = (0,1), \\ u(0) &= u'(1) = 0, \end{aligned} \tag{3.2}$$

where the Dirichlet boundary condition is called an essential boundary condition while the Neumann boundary condition is called a natural boundary condition. The solution to (3.2) is called the solution of the strong form. After multiplying both sides by a test function, $v(x)$, taking the integral of both sides and integrating by parts, we define the associated bilinear form,

$$a(u, v) = \int_0^1 u'(x)v'(x)dx.$$

Considering the space $V = \{v \in L^2([0,1]) : a(v,v) < \infty \text{ and } v(0) = 0\}$, we define the weak form as finding $u \in V$ such that $a(u, v) = \langle f, v\rangle = \int_0^1 f(x)v(x)dx$, $\forall v \in V$. The solution of the strong form is always a solution of the weak form. The converse is only true if $f(x)$ is continuous in Ω and $u \in V \cap C^2([0,1])$.

For the finite-element method, we restrict the solution of the weak form to a finite-dimensional subspace, $S \subset V$, where we define $u_s \in S$ to be the (unique) solution such that $a(u_s, v) = \langle f, v\rangle$, $\forall v \in S$. This is called the Ritz-Galerkin approximation. Indeed, a unique solution exists for the Ritz-Galerkin approximation if $f(x) \in L^2([0,1])$. This solution, u_s, turns out to be the best possible approximation to the solution of the

weak form if measured in energy norm [31],

$$||u||_e^2 = a(u,u).$$

To quantify the quality of the FEM solution, we need to consider a specific choice of the finite-dimensional approximation space, S. To do this, we divide the domain into cells called elements. In one dimension, this is done by dividing the domain into a mesh with points $\{x_j\}_{j=0}^n$, giving elements $[x_{j-1}, x_j]$, with mesh sizes $h_j = x_j - x_{j-1}$. The points between the elements are called nodes.

Within each element, the values of a function are determined by interpolating values at the nodes. So, the finite element method reduces to finding an approximation at each point in the domain, i.e., the nodes. The values at the nodal points are found explicitly and values at the non-nodal points are approximated by interpolation from the nodal values [24].

We then choose a piecewise polynomial representation of u_s on each element. We define the spaces

$$V_k^h = \{u \in V \cap C^0([0,1]) : u(x) \text{ is a polynomial of degree at most } k \text{ for } x \in [x_{i-1}, x_i], \forall i\},$$

and a suitable basis for these spaces. The nodal basis functions for $1 \leq j \leq n-1$ of V_1^k are:

$$\phi_j(x) = \begin{cases} \frac{x - x_{j-1}}{x_j - x_{j-1}}, \text{ for } x_{j-1} \leq x \leq x_j \\ \frac{x_{j+1} - x}{x_{j+1} - x_j}, \text{ for } x_j \leq x \leq x_{j+1} \\ 0, \text{ otherwise.} \end{cases}$$

Basis functions associated with the end points are

$$\phi_0(x) = \begin{cases} \frac{x_1 - x}{x_1 - x_0}, \text{ for } x_0 \leq x \leq x_1 \\ 0, \text{ otherwise,} \end{cases}$$

$$\phi_n(x) = \begin{cases} \frac{x - x_{n-1}}{x_n - x_{n-1}}, \text{ for } x_{n-1} \leq x \leq x_n \\ 0, \text{ otherwise.} \end{cases}$$

Note that these functions satisfy $\phi_j(x_i) = 1$ if $i = j$ and zero if $i \neq j$. The nodal functions above are linear polynomials and are graphed in Figure 3.2.

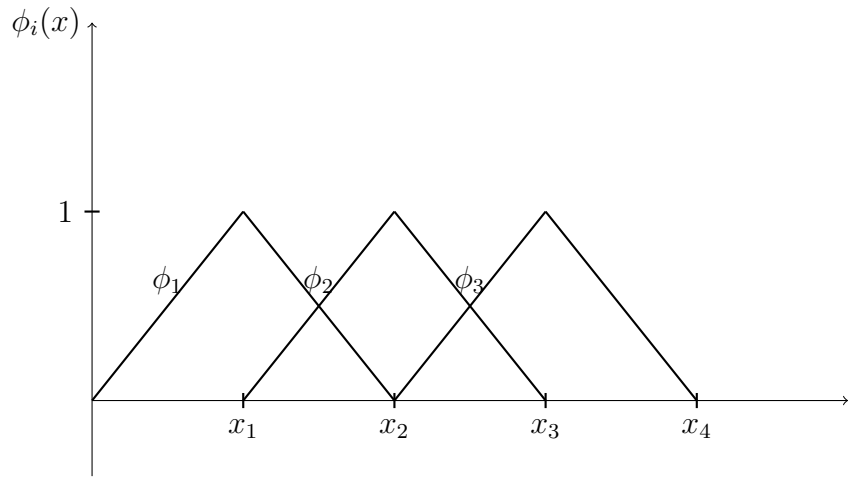


Figure 3.2: Basis functions of 1D FEM are piecewise polynomial functions.

Notice that ϕ_0 is not included in Figure 3.2 nor in the expansions that follow. This is because ϕ_0 is the basis function at the end point, $x_0 = 0$, and the value of $u(0)$ is a fixed known value, which is 0.

For a function $u_h \in V_k^h$, the weak form specifies $\int_0^1 u_h' v_h' dx = \int_0^1 f(x) v_h dx$, $\forall v_h \in V_k^h$. Writing $u_h = \sum_{j=1}^n u_j \phi_j$, this becomes $\sum_{j=1}^n u_j \int_0^1 \phi_i' \phi_j' dx = \int_0^1 f(x)\phi_i dx$, for $1 \leq i \leq n$ which can be written as $Au = b$, where

$$u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_n \end{bmatrix} \text{ and } b = \begin{bmatrix} \int_0^1 f(x)\phi_1 dx \\ \int_0^1 f(x)\phi_2 dx \\ \vdots \\ \vdots \\ \int_0^1 f(x)\phi_n dx \end{bmatrix}.$$

Matrix A it is called the global stiffness matrix. In this setting, the matrix is symmetric and tridiagonal. The elements of A are $a_{ij} = a(\phi_i, \phi_j)$ and can be found as:

$$\begin{aligned} a_{11} &= \int_0^1 \phi_1'\phi_1' dx = \int_0^{x_1} \left(\tfrac{1}{h_1}\right)^2 dx + \int_{x_1}^{x_2} \left(\tfrac{-1}{h_2}\right)^2 dx \\ a_{12} &= \int_0^1 \phi_1'\phi_2' dx = \int_{x_1}^{x_2} \tfrac{-1}{h_2^2} dx \\ a_{22} &= \int_0^1 \phi_2'\phi_2' dx = \int_{x_1}^{x_2} \left(\tfrac{1}{h_2}\right)^2 dx + \int_{x_2}^{x_3} \left(\tfrac{-1}{h_3}\right)^2 dx, \\ \vdots &= \vdots \end{aligned}$$

Note $a_{ij} = a_{ji}$. In general,

$$\begin{aligned} a(\phi_{i-1}, \phi_i) &= \int_{x_{i-1}}^{x_i} \phi_{i-1}'(x)\phi_i'(x) dx \\ a(\phi_i, \phi_i) &= \int_{x_{i-1}}^{x_i} (\phi_i'(x))^2 dx + \int_{x_i}^{x_{i+1}} (\phi_i'(x))^2 dx \\ a(\phi_{i+1}, \phi_i) &= \int_{x_i}^{x_{i+1}} \phi_{i+1}'(x)\phi_i'(x) dx. \end{aligned}$$

We note here that $a(\phi_n, \phi_n)$ cannot be defined in the same way as the general case, since the interval $[x_n, x_{n+1}]$ lies outside of the physical domain. Instead, we define $a(\phi_n, \phi_n) = \int_{x_{n-1}}^{x_n} (\phi_n(x))^2 dx$. Note that this is quite different than the treatment of the Dirichlet boundary condition, where ϕ_0 is dropped completely from the discretized form.

A thus becomes

$$\begin{bmatrix} \frac{1}{h_1} + \frac{1}{h_2} & \frac{-1}{h_2} & 0 & \cdots & & & \\ \frac{-1}{h_2} & \frac{1}{h_2} + \frac{1}{h_3} & \frac{-1}{h_3} & 0 & \cdots & & \\ 0 & \ddots & \ddots & & & & \\ & & & \ddots & \ddots & \frac{-1}{h_{n-1}} & 0 \\ & & & & \frac{-1}{h_{n-1}} & \frac{1}{h_{n-1}} + \frac{1}{h_n} & \frac{-1}{h_n} \\ & & & & & \frac{-1}{h_n} & \frac{1}{h_n} \end{bmatrix}.$$

We note that in finite element methods, boundary conditions, both essential and natural, are accounted for. Natural boundary conditions are implicitly included in the method by their use in the integration by parts step when defining the weak form. This is seen using Green's theorem. This term drops out in our problem because the natural condition is chosen to be homogeneous. If the natural condition is

non-homogeneous, an adjustment to the construction of b is required such that the boundary condition is accounted for in the last entry. Dealing with Neumann conditions is easier in FEM than in finite difference methods. This is why these conditions are called natural conditions. As for the essential conditions, they are explicitly imposed by requiring both the test function v to satisfy $v(0) = 0$, and requiring the Ritz-Galerkin approximation to satisfy $u_s(0) = 0$.

In summary, the essential steps of discretizing using finite elements are:

1. Formation of the weak form of the boundary value problem.
2. Partition of domain into elements, called triangulation.
3. Define finite element space, V_k^h.
4. Build the basis functions of the space in step 3.
5. Assemble the global stiffness matrix and solve the finite-element system $Au = f$.

In two dimensions, the finite-element method is slightly more complicated. The elements are usually triangular or quadrilateral, in contrast to the elements in 1 dimension [31]. In both cases, the process of forming these elements is usually called triangulation. A mesh of triangular shaped elements is shown in Figure 3.3. Each element shares 2 nodes and an edge with each adjacent element. The vertices of the triangles are the nodes denoted as (x_i, y_i). More approximation nodes can be added in the middle of the edges for higher-order 2D finite-element methods.

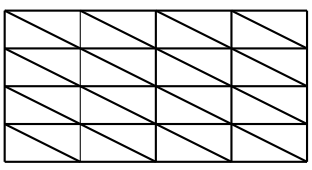


Figure 3.3: Triangular finite elements on 2D mesh.

Again, we need a space V_k^h with a piecewise polynomial basis such that

$$V_k^h = \{u \in C^0(\Omega) : \forall\ T \in \Omega^h, u(x) \text{ is a polynomial of degree no more than k on } T\}.$$

Basis functions can be defined by polynomial interpolation of nodal values. So, the basis functions ϕ_i, for $i = 1, 2, 3$ in a triangle, T, associated with a point (x_i, y_i) can be written as

$$\phi_i(x, y) = a_i + b_i x + c_i y,$$

where a_i, b_i, c_i are constants to be found later. Since the element, T, is triangular, we find ϕ_1, ϕ_2 and ϕ_3 for the three different nodes. Taking the vector $[1, x, y]$ to be the vector of polynomial basis functions, $\phi_i(x, y)$ can be written in matrix form as

$$\phi_i(x, y) = \begin{bmatrix} 1 & x & y \end{bmatrix} \begin{bmatrix} a_i \\ b_i \\ c_i \end{bmatrix}.$$

We then require

$$\phi_i(x_j, y_j) = \begin{cases} 1, \text{ for } i = j \\ 0, \text{ for } i \neq j \end{cases}$$

Thus, $\phi_1(x_1, y_1) = 1$, $\phi_1(x_2, y_2) = 0$ and $\phi_1(x_3, y_3) = 0$, with similar requirements for ϕ_2 and ϕ_3. So, for the first basis function, we have the system of equations:

$$\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

By Gaussian elimination, the system can be solved to find the values a_1, b_1, c_1. These are

$$\begin{aligned} a_1 &= \frac{x_2 y_3 - y_2 x_3}{(x_2 y_3 - x_3 y_2) + (y_2 - y_3) x_1 + (x_3 - x_2) y_1} \\ b_1 &= \frac{y_2 - y_3}{(x_2 y_3 - x_3 y_2) + (y_2 - y_3) x_1 + (x_3 - x_2) y_1} \\ c_1 &= \frac{x_3 - x_2}{(x_2 y_3 - x_3 y_2) + (y_2 - y_3) x_1 + (x_3 - x_2) y_1}. \end{aligned}$$

The final step is to assemble the stiffness matrix. This is done in 2 steps, defining the

element stiffness matrix and then assembling the global stiffness matrix.

The element matrix for an element T, denoted as A^T, is formed by $a_T(\phi_k, \phi_j) = \iint_T \nabla\phi_j \cdot \nabla\phi_k dA$. Consider the first element shown in Figure 3.4.

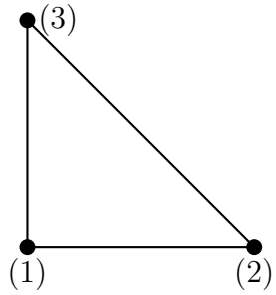


Figure 3.4: A single triangular element T_1.

If node (1) has coordinates of (x_i, y_j), then (2) has coordinates of (x_{i+1}, y_j), and (3) has coordinates of (x_i, y_{j+1}). As above, we can find the basis functions for the three nodes,

$$\begin{aligned}
\phi_1(x,y) &= \tfrac{(y_{i+1}-y)}{(y_{i+1}-y_i)} + \tfrac{(x_i-x)}{(x_{i+1}-x_i)} \\
\phi_2(x,y) &= \tfrac{-x_i}{x_{i+1}-x_i} + \tfrac{x}{x_{i+1}-x_i} \\
\phi_3(x,y) &= \tfrac{-y_i}{y_{i+1}-y_i} + \tfrac{y}{y_{i+1}-y_i}.
\end{aligned}$$

We then find the vectors $\nabla\phi_1, \nabla\phi_2, \nabla\phi_3$ and then integrate $a_T(\phi_k, \phi_j)$ over the element T to find all entries of the element stiffness matrix A^T.

We then extend all element matrices to a matrix of the size $n \times n$, where n is the total number of nodes in the mesh. Entries of A^T are filled in to the bigger matrix at places corresponding to the nodes associated with each element. After this extension, the global stiffness matrix is equal to the summation of the extended element stiffness matrices.

3.2 Solution of Stochastic Model

We need to find a method that will give us a good approximate solution for (3.1). This problem can be discretized using FEM by multiplying both sides with v and integrating both sides to get

$$-\int_\Omega \nabla \cdot (K\nabla u)v = \int_\Omega fv, \ \forall v.$$

Integrating by parts, we have

$$\int_\Omega (K\nabla u) \cdot \nabla v = \int_\Omega fv, \ \forall v. \tag{3.3}$$

Now, letting $u = \sum_j u_j\phi_j$, and $v = \phi_i$, (3.3) becomes

$$\sum_j u_j \int_\Omega (K\nabla \phi_j) \cdot \nabla \phi_i = \int_\Omega f\phi_i, \ \forall i.$$

K is uniquely defined at each node, and interpolates over each element adjacent to the node. So, the above equation can be evaluated elementwise, leading to the element stiffness matrices analogous to those discussed above. Note that the continuous system in (3.1) is singular, since both the differential equation and boundary conditions allow any constant shift of a solution, $u(x, y)$, to also be a solution of (3.1). This remains true for the discretized system, $Ax = b$, resulting from this discretization, with the constant vector in the null-space of A. Note, also, that the system is consistent, since the right-hand side vector will be orthogonal to this null-space. Thus, in what follows, we consider solution of the discrete system up to its null-space component (which, if needed, we will fix by taking x to be orthogonal to the constant vector). All of the preconditioners considered below are "robust" to this singularity, perhaps with the addition of a simple orthogonal projection to remove the null-space component of the solution.

To solve this discretized problem, we follow a simple idea of trying to solve (3.1) for many realizations of the coefficient K, using the PCG method along with various preconditioners. A standard choice to use here is AMG, for which we use the

HYPRE BoomerAMG implementation [20, 15]. HYPRE is a library of preconditioners implemented using MPI (message passing interface), which is essential in parallel programming. All of our codes are run in serial, but we use HYPRE for simplicity of implementation.

In this thesis, three algorithms for solving the discretized system in the form $Ax = b$ are considered. As mentioned earlier, these algorithms are all implemented in the C++ language and use FEniCS [49] for the finite-element discretization and PETSc [50] for the numerical solution, possibly interfacing to HYPRE/BoomerAMG. FEniCS is a package like PETSc that provides the user with many commands for solving a differential equation using the FEM method. In the following, four algorithms are discussed with specific parts of codes presented.

As a baseline, we would like to see how efficient it is using a standard preconditioner to solve each realization. To measure efficiency, we monitor the time it takes to converge and the number of iterations the method uses to converge. For this routine, we ran PCG with the AMG preconditioner to solve the discretized problem for 100 realizations of the coefficient field, K, asking for a reduction of the norm of residual by a factor of 10^{-8}. We set the stopping criterion for AMG to be the relative tolerance. For the initial guess of PCG, we use the zero vector. Now, since AMG is a multigrid method, we need to fix a few settings for the routine. These are:

1. **Type of cycle used:** V-cycle.
2. **Maximum number of levels used is:** 25.
3. **Relaxation method:** Jacobi, both for pre- and post-relaxation. Direct solve was used on the coarsest level by simple Gaussian elimination.
4. **Number of sweeps on each level:** 1 both pre and post-relaxation, i.e $m_1 = m_2 = 1$.

Each solve of the problem has its own AMG set-up phase with some cost plus the solve cost. In all our routines implemented in this work, we include several read/write commands from various files to load coefficient values or save solutions, for example. This cost is disregarded in all of our results. Also, some overhead costs were recorded,

consistently in all results, coming from the building of the mesh for the discretized problem and other aspects. We used FEniCS to build these meshes and to define the source of the problem, $f(x, y)$, the variational spaces and the bilinear forms of the FEM method.

As first data, we applied AMG-preconditioned CG to each of 100 linear systems, for each mesh size given in Table 3.1, recording the observed median number of PCG iterations for each solve, as well as the total time to solve all 100 systems.

Mesh size (n)	Iteration count	Total time (in sec.)
16	6	11.3
32	7	15
64	7	29.6
128	8	97.1

Table 3.1: Iteration count and total time of solve AMG precondtioner.

We see here that iteration counts are low, but solve times are substantial, especially on finer meshes. Total time here is the time recorded for the actual solve of the problem. The increase in time as we move to finer meshes is expected because of the increase in the total number of nodes used and the problem sizes to be solved. Noting the large time required for small n, we approximate the time in the above as: time $= c_0 + c_1\left(\frac{n}{16}\right)^2$, where c_0 and c_1 are constants. This yields the system

$$\begin{aligned} 11 &= c_0 + c_1 \\ 15 &= c_0 + 4c_1 \\ 30 &= c_0 + 16c_1 \\ 97 &= c_0 + 64c_1. \end{aligned}$$

Solving this system, we get $c_0 \approx 10$ and $c_1 \approx 1.25$ which (roughly) satisfies all equations. Using finer-grained timings, we see the time associated with c_0 comes primarily from building the mesh and function space in FEniCS. This cost is considered as an overhead that we cannot decrease and have no control over. Dividing c_0 by 100, we find that it takes about 0.1 seconds for FEniCS to build the mesh and function space for each system. This time is seen to be independent of the mesh size, i.e. the number

of nodes used. On the other hand, c_1 is the cost of the algorithm to solve all 100 realizations, on the 16×16 mesh. This is a reasonable method with reasonable times, but the focus of this work is whether we can decrease these times by incorporating a better solution method.

AMG is an effective "black box" to solve these linear systems, but has no ability to incorporate information emerging from the solves. Deflation, in contrast, is typically less effective as a preconditioner, but has low cost per iteration, and information can be easily added into the deflation space. We adapt our deflation space here by following two separate methods. In both, we divide our set of permeability fields into two subsets, called the training set and testing set. The training set is usually a small set of coefficients, ranging from 2 to 16 realizations. The testing set is composed of the coefficient realizations remaining from the 100 selected, after removing those from the training set. Information obtained using the training set is used to help solve the problems in the testing set. The two methods proposed here both use the singular value decomposition (SVD). These methods, in brief, are:

Solution-based deflation: Solve the systems $A_i x_i = b$ for the training set to get a set of solution vectors $X = [x_1, x_2, \ldots, x_m]$. Then, find the SVD of X and identify a few of the vectors with largest singular values. Finally, solve the remaining systems $A_i x_i = b$ for the testing set, using these singular vectors to define the deflation space.

Eigenvector-based deflation: For the systems $A_i x_i = b$ in the training set, find t of the smallest eigenvalues of A_i and their associated eigenvectors,
$E = [e_{11}, e_{21}, \ldots, e_{t1}, \ldots, e_{1m}, \ldots, e_{tm}]$. Then, find the SVD of E and identify a few of the vectors with largest singular values. Finally, solve $A_i x_i = b$ for the testing set, using these singular vectors to define the deflation preconditioner.

The rest of this chapter focuses on our work on the two methods above. We start by describing solution-based deflation in detail. This method consists of 3 parts implemented by 3 individual codes. We note that this method has some similarity to reduced-basis methods [13]; however, here, we use the reduced-order model as a component of our preconditioner, but still solve the full fine-scale discretization.

We start by solving the discretized systems $A_i x_i = b$ for the training set just as above. For this part, we run the same routine as before using AMG from the HYPRE library with the same parameters as set previously. For each coefficient in the training set, we form A_i, and solve using AMG-preconditioned CG, giving an approximate solution denoted as x_i. These solution vectors are stored column-wise in a matrix denoted as X, such that $X = [x_1, x_2, \ldots, x_m]$. The number of iterations performed is consistent with those reported above, with about 8 iterations per solve. Time recorded for this part of this method includes:

1. Assembly of mesh and finite element space using FEniCS.
2. Defining the bilinear form, assembly of A, setting the parameters of AMG and solving the system.

Condensed code to achieve this is given below.

```
// Define variational forms
Poisson1::BilinearForm a(V1, V1);
Poisson1::LinearForm L(V1);
auto f = std::make_shared<Source>();
auto g = std::make_shared<dUdN>();
auto k = std::make_shared<dolfin::Function>(V1,coefvec);
L.f = f;
L.g = g;
a.k = k;

//Assemble the system
auto A=std::make_shared<PETScMatrix>();
PETScVector b;
assemble(*A,a);
assemble(b,L);

//solve
solver.solve(*u.vector(), b);
```

In the above, the definition of variational forms is contained in a "UFL form file". Source is a function defining the right-hand side of (3.1), dudN is a function defining

the (homogeneous) Neumann boundary condition, and coefvec is a FEniCS vector whose elements are the values of the coefficient K at each node on the fine mesh. The bilinear form, denoted a, using the finite-element space denoted as V_1^h above is

$$a(u,v) = \int_\Omega (K\nabla u)\cdot(\nabla v)dx.$$

The linear form, denoted by L, is

$$L(v) = \int_\Omega fvdx.$$

The associated weak form is to find $u \in V_1^h$ such that

$$a(u,v) = L(v), \text{ for } v \in V_1^h.$$

A is the matrix assembled from the bilinear form $a(u,v)$, and b is the vector defined by the linear form $L(v)$. Both are assembled using data types from the PETSc library. This system is solved by the solver.solve command with options set via the PETSc interface.

We now move on to the second part, which includes finding the SVD of X. In the singular value decomposition, the given matrix X of size $n \times p$ is represented by the product of matrices, USV^T, where S is the same dimension as X and the diagonal elements of S are the singular values, while all other values in S are zero. The columns of the $n \times n$ matrix U are called the left singular vectors while the columns of the $p \times p$ matrix V are called the right singular vectors. Both U and V are orthogonal matrices.

For this second part of the method, PETSc has a very useful extension called SLEPc [51]. Using the following clip of code, we obtain the singular vectors.

```
1 //find the svd for X_mat
2 SVD svd;
3 PetscInt its,nsv,ncov,ncv;
4 ncv=50;
5 nsv=25;
6 PetscReal error,sigma;
```

```
Vec uu,vv;
MatCreateVecs(A,&uu,&vv);
ierr=SVDCreate(PETSC_COMM_WORLD,&svd);
ierr=SVDSetOperator(svd,X_mat);
ierr=SVDSetType(svd,SVDCROSS);
ierr=SVDSetFromOptions(svd);
ierr=SVDSolve(svd);
ierr=SVDGetIterationNumber(svd,&its);
```

Here, *svd* is the SLEPc SVD object. We set $nsv = 25$ to request 25 singular values be approximated and $ncv = 50$ to be the maximum dimension of the subspace used by the solver. We initialize *svd* and set X_mat to be the associated matrix of which the SVD is computed. SVDSolve(svd) is the command to solve the SVD problem. The total number of iterations executed in the SVD process is saved in its.

The singular vectors in U, ordered from largest singular value to smallest, provide a basis for the space generated by the span of X. Those associated with the largest singular values are, in some sense, most prominent in the solution set. Assuming the solution character does not change between the training and the testing sets, we use these to define deflation vectors for use in preconditioning the testing set.

Singular vectors associated with the largest singular values are produced and saved to file. Since no linear system solve is performed here, the only time taken into consideration is that of the SVD solver finding the singular values and vectors. We note here the time of this part is usually the smallest contributing time to the overall total time of the method. An added purpose of using the SVD is to insure that the set of vectors used in the deflation matrix are linearly independent, so that the projected system, E, is non singular (aside, perhaps, from inheriting the singularity of A).

After these vectors have been stored, they are then read into the final stage, which is defining a deflation preconditioner and solving the systems $A_i x_i = b$ for the testing set. In this routine, we construct a deflation matrix denoted as Z. This matrix consists of columns that are the singular vectors found previously. The size of Z depends on the mesh size and number of singular values taken. For example, if we were to take only 4 singular vectors, then the size of Z would be $(n^2 \times 4)$.

As a further step, we also consider results when Z is formed using subdomains, i.e. that the domain is divided into several equally sized geometric ranges. For example, for 4 subdomains we could divide Ω into:
1- $x \in [0, 1.5)$ and $y \in [0, .5)$,
2- $x \in [1.5, 3]$ and $y \in [0, .5)$,
3- $x \in [0, 1.5)$ and $y \in [.5, 1]$,
4- $x \in [1.5, 3]$ and $y \in [.5, 1]$.

The subdivisions of Ω into 4, 8, 12, and 16 subdomains are illustrated in Figure 3.5. There are, of course, many ways of dividing Ω into geometric subdomains. For instance, 8 subdomains can naturally be either a 4×2 or 2×4 array. We disregard the effect of this as a parameter in our research.

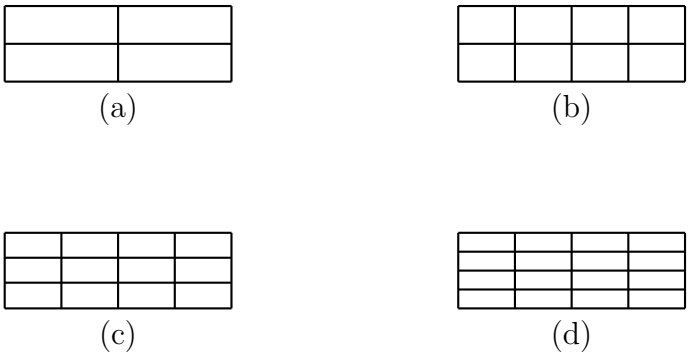


Figure 3.5: Ω divided into subdomains:(a) 4 subdomains, (b) 8 subdomains, (c) 12 subdomains, (d) 16 subdomains.

Note there is no overlapping of subdomains, and nodes on the geometric boundary between two subdomains are included in only 1 subdomain. For each subdomain and each vector from the SVD, there is an associated deflation vector (column) in Z. In other words, each singular vector is divided up over these subdomains. So for the deflation vector, v, on the subdomain D, corresponding to SVD vector Z,

$$v = \begin{cases} Z(x_i, y_j), \text{ if } (x_i, y_j) \in D \\ 0, \text{ otherwise.} \end{cases}$$

Each singular vector is divided into deflation vectors that correspond to each subdomain. The advantages of deflation have been discussed in the previous chapter; here, we add that subdomains are used because we expect convergence to be faster with a preconditioner using subdomains than that with a deflation preconditioner with no subdomain division. This is due to both the sparsity structure of this new Z and the increasing dimension of span(Z).

Numerical tests show that in order to construct an effective deflation space, we have to take the constant vector, **1**, divided over subdomains, and add the resulting vectors as columns of Z. From the original PDE, we know that since the gradient of a constant function is zero, then **1** is in the "near null space" of the discretization matrices, A_i. **1** is always associated with the slowest converging error, so if **1** is not included in the construction of Z, then convergence may be very poor.

```
Mat PP;
int col;
double val[1];
ierr=MatCreateSeqAIJ(PETSC_COMM_WORLD,(nx*ny),ncol,5,NULL,&PP);
MatSetOption(PP,MAT_NEW_NONZERO_LOCATION_ERR,PETSC_FALSE);
MatSetOption(PP,MAT_IGNORE_ZERO_ENTRIES,PETSC_TRUE);
ierr=MatSetUp(PP);
lx=3/subx; ly=1/suby;

int i=0,j=0;
for(c=0;c<subx*suby;c++)
{if(j<suby)
 {if(i<subx)
    {    col=c;
     for(int q=0;q<nx*ny;q++)
      {if( (Vdofcoords[2*q]>=(i*lx)) && (Vdofcoords[2*q]<=((i+1)*lx)) &&
    (Vdofcoords[2*q+1]>=(j*ly)) &&  (Vdofcoords[2*q+1]<=((j+1)*ly))   )
           {val[0]=svd_sol1_new[q];
           ierr=MatSetValues(PP,1,&q,1,&col,val,INSERT_VALUES);}
      }
   i++;
  }
  else
  { i=0;j++;c  ;}
```

```
24 }
25 else
26 printf("0\n");
27 }
```

Here, *subx* and *suby* are the user-defined values denoting the number of subdomains taken in the x-direction and y-direction, respectively. *col* is a specific column number of the deflation matrix, which is denoted by PP, and *ncol* is the total column size of PP, such that *ncol* is the number of subdomains taken multplied by the number of vectors to be divided over the subdomains, i.e, $ncol = no\_subdomains * (total\_svd + 1)$. Thus, the dimension of PP is $(nx * ny) \times ncol$. The MatSetOption commands indicate to set PP as a sparse matrix. For each svd vector, we loop over all subdomains in the x-direction then y-direction (lexicographical ordering), dividing the vector into multiple vectors with entries equalling the value of the particular svd vector under consideration at a node that lies in that subdomain and 0 otherwise. These subdomain vectors are stored column-wise in PP. We emphasize here that the use of node locations is important, since we do not order our vectors geometrically.

Having defined the deflation matrix, Z, we now solve the linear systems in the testing set using deflation-preconditioned CG. For convenience, we implement this as a balancing preconditioner, using a composite preconditioner in PETSc. The deflation solve itself is implemented as a Galerkin preconditioner, consisting of a two-level numerical process with interpolation matrix as the deflation matrix, Z, discussed above, either with subdomains or without, and restriction as Z^T. As a direct solver on the coarsest grid, with $E = Z^T AZ$, an LU decomposition method was used. This is complemented by an ILU preconditioner, for which we use the standard incomplete LU factorization of A used with no fill-in. For details on ILU, see [38].

The balancing preconditioner is a composite preconditioner with three stages, corresponding to a Galerkin method, then an ILU, which we denote by M, and then Galerkin again. The residual from the CG method is input into the first Galerkin preconditioner. A deflated PCG method is applied where the residual is restricted to the coarse level, the coarse-level system is solved by a direct solver and a correction is interpolated to the fine grid. The process, following the PCG algorithm in Chapter

2, is seen as first solving for x_c in

$$Ex_c = Z^T r^{(k)},$$

where E is the Galerkin matrix defined in deflated PCG earlier, and $r^{(k)}$ is the residual associated with the current CG iteration. Then, the coarse level solve and interpolation gives

$$\delta x^{(G,1)} = Zx_c = ZE^{-1}Z^T r^{(k)}.$$

The corrected residual is then passed on to the second preconditioner, ILU. Denoting the incomplete factorization as M, the system

$$M\delta x^{(F)} = r^{(k)} - A\delta x^{(G,1)},$$

is solved for the solution vector of this method on the finest grid, $\delta x^{(F)}$. Note that the right hand side includes the correction from the first step.

Finally, the residual of that system is then passed on to the last preconditioner, where a Galerkin method is again applied solving

$$Ex_c = Z^T(r^{(k)} - A\delta x^{(G,1)} - A\delta x^{(F)}).$$

Then, the solution vector for the second Galerkin method, $x^{(G,2)}$ is found by

$$\delta x^{(G,2)} = Z(E^{-1}Z^T(r^{(k)} - A\delta x^{(G,1)} - A\delta x^{(F)})).$$

So, the composition of preconditioners gives in the end a solution of the system $A\delta x = r^{(k)}$ in the form of a summation of $\delta x^{(G,1)}$, $\delta x^{(F)}$, $\delta x^{(G,2)}$. Code to define this in PETSc is given as

```
1 //first pc
2 PC P1;
3 PC coarse_pc;
4 ierr=PCCompositeAddPC(pc,PCGALERKIN);
```

```
ierr=PCCompositeGetPC(pc,0,&P1);
ierr=PCGalerkinSetRestriction(P1,PP);
KSP ksp_gal;
ierr=PCGalerkinGetKSP(P1,&ksp_gal);
Mat A_galerkin;
ierr = MatRARt(A,R,MAT_INITIAL_MATRIX,PETSC_DEFAULT,&A_galerkin);
ierr=KSPSetOperators(ksp_gal,A_galerkin,A_galerkin);
KSPGetPC(ksp_gal,&coarse_pc);
PCSetType(coarse_pc,PCLU);

//second pc
PC P2;
ierr=PCCompositeAddPC(pc,PCILU);
ierr=PCCompositeGetPC(pc,1,&P2);
ierr=PCSetType(P2,PCILU);

//third pc
PC P3;
PC coarse_pc2;
ierr=PCCompositeAddPC(pc,PCGALERKIN);
ierr=PCCompositeGetPC(pc,2,&P3);
ierr=PCGalerkinSetRestriction(P3,PP);
KSP ksp_gal_2;
ierr=PCGalerkinGetKSP(P3,&ksp_gal_2);
Mat AA_galerkin;
ierr = MatRARt(A,R,MAT_INITIAL_MATRIX,PETSC_DEFAULT,&AA_galerkin);
ierr=KSPSetOperators(ksp_gal_2,AA_galerkin,AA_galerkin);
KSPGetPC(ksp_gal,&coarse_pc2);
PCSetType(coarse_pc2,PCLU);
```

In the algorithm presented above, PC1 is the first Galerkin preconditioner. PCCompositeADDPC is the command to add a new preconditioner to the multiplicative process such that PCCompositeGetPC(pc,0,&P1) just sets P1 to be the first preconditioner. We set the deflation matrix, PP, to be the restriction matrix in the Galerkin method. It is enough in the PETSc routines to indicate either the restriction matrix or the interpolation matrix since, by default, one is the transpose of the other. *A_galerkin* denotes the product matrix solved in the Galerkin method i.e $A\_galerkin = (PP)^T A(PP)$ and is set using the command MatRARt. $(PP)^T$ is denoted by R in our algorithm, such that hence $A\_galerkin = RAR^T$, and *coarse_pc* and *coarse_pc*2 are both the LU direct solve preconditioner in PC1 and PC3, receptively.

The second preconditioner is denoted as P2 and set to be the incomplete factorization.

Within this methodology we identify three important parameters (with an additional fourth in the next approach):

1. Number of problems in the training set.
2. Number of singular vectors used in deflation.
3. Number of subdomains considered.

In the following chapter, we experiment with finding optimal values of these parameters.

For the eigenvector-based deflation approach, instead of using solutions of the problems in the training set, we find the eigenvectors associated with the t smallest eigenvalues of each matrix in the training set. We consider $t \leq 15$, for each system. The respective eigenvectors, e_{ik} for $i = 1, 2, \ldots, t$, $k = 1, 2, \ldots, m$ are stored column-wise in a matrix denoted as E, such that $E = [e_{11}, e_{21}, \ldots, e_{t1}, \ldots, e_{1m}, \ldots, e_{tm}]$. Note this matrix replaces matrix X in the previous method. We consider taking eigenvectors instead of solutions in this part, because the solution of (3.1) is dominated by the eigenvectors associated with the smallest eigenvalues. This is since the eigenvector expansion of any SPD linear system, $Ax = b$, can be written as

$$Ax = b = \sum_i c_i v^{(i)},$$

where c_i is determined by the right-hand side, b, and $v^{(i)}$ is the i^{th} eigenvector of A. Assuming $Av^{(i)} = \lambda_i v^{(i)}$ and $\lambda_i \neq 0$, then we can write the solution as

$$x = \sum_i \frac{c_i}{\lambda_i} v^{(i)}.$$

So, since λ_i is in the denominator, the smaller the eigenvalue, the larger the contribution to the solution, if all c_i are roughly the same size.

The code presented in the following clarifies the eigenvalue computation.

```
//find the least eigenvector for Ak
Mat AA=A->mat();
PetscViewer viewer;
EPS eps;
PetscInt nev, max_its;//number of required iterations ie eigenvalues
nev=10;
max_its = 1000;
Vec xr,xi;
PetscScalar eigenreal;
PetscScalar eigencomplex;
PetscInt its,nconv;
ST st;

ierr=MatCreateVecs(AA,&xr,&xi);
ierr=EPSCreate(PETSC_COMM_WORLD,&eps);
ierr=EPSSetOperators(eps,AA,NULL);
ierr=EPSSetProblemType(eps,EPS_HEP);
ierr=EPSSetWhichEigenpairs(eps,EPS_SMALLEST_REAL);

ierr=EPSSetDimensions(eps,nev,max_its,PETSC_DEFAULT);
ierr=EPSSetFromOptions(eps);
ierr=EPSSolve(eps);
```

In the above code, *eps* is the SLEPc object for its eigenvalue solver, *nev* is the number of requested eigenvalues of the problem, and *max_its* is the maximum dimension of the subspace associated with the eigenvalue problem. We define xr to be the real component of the eigenvector e_{ik}, requesting only the eigenvectors associated with the smallest eigenvalues with the command line EPSSetWhichEigenpairs(eps, EPS_SMALLEST_REAL). The type of eigenvalue problem is set here with EPS_HEP, which indicates that A is Hermitian.

We run this code with m problems in the training set and find t eigenvectors for each matrix. Next, we use the eigenvectors found in the previous part to define the deflation space. This is done using the exact same process as in the second part of the previous method. We also use the same code to execute this. Singular vectors are found and stored ready for use in the last part of the method.

The last part of this method is also the same as the the first method. We also use deflation, with the same three preconditioners and balancing technique. In this approach, as above, we also identify important parameters to test. These are the three mentioned earlier, plus the number of eigenvectors taken for each problem in the training set. Since these replace the solution vectors in defining the deflation space in the previous approach, we expect variation in the number of eigenvectors considered to have some effect in the overall performance of this method.

Chapter 4

Results/Conclusions

In this chapter, we present the results that were obtained from the algorithms proposed above, and compare them to results found using the AMG-preconditioned CG algorithm. At the end of this chapter, we conclude our results and discuss possibilities for future work.

For both methods proposed above for defining the deflation matrix and balancing preconditioner, we aim to see if we can achieve better efficiency, in terms of iteration counts and total run-times, than simply solving the systems with AMG-preconditioned CG.

4.1 Results for Solution-based deflation:

Recall, we have 3 main parameters here to consider, number of problems in the training set, number of singular vectors taken and number of subdomains considered. We go through these parameters fixing all but one, varying each in order to find the best value for that parameter.

Before we examine the different parameters, we were curious to see in detail the time required for solution of the problems in the testing set using the balancing preconditioned CG algorithm, to identify which component of this part takes up most

of the computing time. We recorded timing for the four main aspects of the code related to solving $Ax = b$, divided as follows:

1. Time 1: time required for FEniCS to assemble the mesh on the finest level and to define the function space V_1^h.
2. Time 2: time spent by FEniCS to define the bilinear and linear forms.
3. Time 3: time required by FEniCS to assemble the matrix A defined in the discretized problem in PETSc format.
4. Time 4: PETSc time to define the three preconditioner matrices and PETSc solve time.

For Table 4.1, we fixed the number of problems in the training set to be 13, the number of singular vectors used to define the deflation preconditioner to be 4, and the number of subdomians to be 4, where the number of subdomains in both the x and y-directions is 2. We note that we add the constant vector, $\mathbf{1}$, to the set of singular vectors so the number of columns in matrix P is 20. All times in this chapter are in seconds.

Mesh size ($n \times n$)	Time 1	Time 2	Time 3	Time 4
16^2	7.9	0.0014	0.11	0.8
32^2	8.15	0.0014	0.3	1.9
64^2	8.0	0.015	1.04	8.13
128^2	6.6	0.018	4.5	78.24

Table 4.1: Detailed timing of solution of testing set for Solution-based deflation with 13 problems in the training set, 4 singular vectors used to define the deflation space, and 4 subdomains.

The first observation from this data is that the generation of meshes and function spaces, Time 1, is roughly independent of the mesh size. Since these times are accumulated from times recorded in all solves with the 87 problems in the testing set, each solve with one of these vectors takes about 0.1 seconds to build a mesh and function space. This could, perhaps, be removed by optimizing code to do this only once for all problems in the testing set, but we have not considered this here, due to practical

difficulties in memory management with FEniCS.

We also notice it is very cheap for FEniCS to define the variational forms in the FEM. The cost that is really of interest is the cost spent on the actual calculation process, Time 4. It is natural that this cost increases with the increase of mesh size of the problem, since the number of nodes used for solving our problem is increased. Assuming perfect scaling, given a solve time of 8 seconds for 64^2 nodes, the solve time for 128^2 nodes would be about 32 seconds, indicating a suboptimal preconditioner is used in those results.

We now move to choosing appropriate parameters. We first fix the training set to have 8 problems and the number of singular vectors used in defining the deflation preconditioner to be 4 and, hence, vary the subdomain count. For simplicity, we use odd-numbered subdomain decompositions on meshes with numbers of elements given by powers of 2 subdomains. This simplifies the implementation of loops in the subdomain code by removing any decisions regarding overlap. For 1 subdomain the mesh is divided as 1×1, for 9, the mesh is divided into 3×3 subdomains, and, for 49 subdomains, the mesh is divided into 7×7 subdomains. The total time required for this method and observed median iteration counts while varying the number of subdomains is recorded in Table 4.2.

nodes	sub=1		sub=9		sub=49	
	its	time	its	time	its	time
16^2	15	10.73	13	10.98	-	-
32^2	28	12.16	16	21.76	-	-
64^2	58	18.80	45	18.81	43	21.4
128^2	120	143.84	51	68.57	25	200.19

Table 4.2: Time and iteration counts (its) for Solution-based deflation with varying numbers of subdomains (sub).

In Table 4.2, it is impractical to use a deflation matrix with 49 subdomains for the smaller sized meshes. We find that the iterations needed to converge consistently decreases when using more subdomains, but that the overall cost increases when too

many subdomains are used. We find that the smallest times are seen when 1 subdomain is used for construction of a deflation space for the 16^2 and 32^2 meshes and 9 subdomains is used for the 64^2 and 128^2 meshes.

Fixing these subdomain counts, we move to variation in size of the training set. We keep the number of singular vectors chosen to define the deflation space fixed to 4 vectors plus the constant vector, $\mathbf{1}$. We test for 4, 8, 12 and 16 problems in the training set. Iteration counts and total time spent by the algorithm are presented in Table 4.3.

nodes	training=4		training=8		training=12		training=16	
	its	time	its	time	its	time	its	time
16^2	15	10.68	15	10.73	15	10.73	15	10.57
32^2	31	12.45	31	12.40	31	12.16	31	12.79
64^2	44	16.31	44	17.43	44	18.81	44	19.77
128^2	51	65.64	51	65.71	51	68.57	51	67.81

Table 4.3: Time and iteration counts (its) for Solution-based deflation with varying number of problems in the training set and 4 singular vectors used to define the deflation space.

Notice that, in Table 4.3, total times recorded increase the as we increase problem size. The smaller the training set, the less time is used in solving these problems using AMG and more time is naturally spent in the solution of the testing set problems, since 4 problems in the training set means there are 96 problems in the testing set, 8 problems in the training set means 92 problems in the testing set, and so on. Notice also that the number of iterations does not change as we increase the size of the training set, which is expected. The important realization to take from this study is that the cost of this algorithm is largely independent of the size of the testing and training sets chosen when keeping the number of vectors taken from the SVD fixed. For convenience in optimizing the last parameter, we fix the size of the training set to be 12. We note that further increasing the size of the training set may improve the performance of the resulting preconditioner, by including more information about the linear systems in the definition of the deflation matrix; however, this also increases the overall cost of solution, since the resulting deflation preconditioner is more efficient than the black-box AMG preconditioner used for the training set. Appropriately

balancing these costs is an important factor when considering a fixed number of total linear systems to be solved.

We finally consider the last parameter, choosing the optimal number of singular vectors taken. We vary this parameter to be 0,1,2,4 and 8, and remind the reader both that this determines the number of columns in the deflation matrix, P, and that the constant vector, $\mathbf{1}$, is always included with these vectors and divided over the subdomains on the mesh. So, 0 singular vectors means that only the constant vector is used in defining P. Results are presented in Table 4.4.

nodes	singular=0		singular=2		singular=4		singular=8	
	its	time	its	time	its	time	its	time
16^2	20	10.76	17	10.72	15	10.73	14	10.86
32^2	40	12.79	34	13.00	31	12.16	25	12.41
64^2	63	20.69	52	21.33	44	18.81	37	15.54
128^2	114	115.34	68	79.33	51	68.57	44	71.57

Table 4.4: Time and iteration counts (its) for Solution-based deflation with varying numbers of singular vectors used to determine the deflation matrix and 12 training vectors.

In Table 4.4, using no singular vectors, the SVD time is, of course, zero since no singular vectors are used in defining the balancing preconditioner. We note that the SVD time in each variation is the same, since SLEPc calculates all singular vectors of the solution matrix, X. We also note that time spent for AMG preconditioned CG is fixed, since the size of the training set is fixed.

From the table above, we see that the iteration count decreases for each mesh as the number of singular vectors chosen is increased. This is attributed to the advantage of using deflation over a larger space. Total time also varies, although irregularly. This comes from the varying size of P in the last part of the algorithm and the balance between cost per iteration and number of iterations. We find that the smallest times per iteration are generally when 4 singular vectors are used.

Overall, the best times are recorded for this method when choosing a training set containing 12 permeability fields, and 4 singular vectors with (1,1,9,9) subdomains for respective mesh sizes. The method spends about 0.11 seconds to solve $Ax = b$ for each permeability field for a 16^2 mesh, about 0.12 seconds per problem for the 32^2 mesh, about 0.19 seconds per problems for the 64^2 mesh, and about 0.69 seconds per problem for the 128^2 mesh.

4.2 Results for Eigenvector-based deflation:

In this section, we present results obtained from following the same process of choosing parameters in the previous section, but for Eigenvector-based deflation. Recall the main difference in these methods is the use of eigenvectors from the training set matrices to find the deflation space, instead of using solution vectors from the training set. This change requires the addition of a new parameter mentioned earlier, the number of eigenvectors taken for each problem in the training set. This parameter is consistent for all problems in the testing set chosen.

We begin with again considering variations in the number of subdomains. For consistency in comparison to other algorithms, we test for the same subdomain counts as in the previous section, i.e 1, 9, and 49. For the finest mesh, 128^2, we also added 255 subdomain results, where we set 15 subdomains in both the x and y directions. In optimizing this parameter, we fixed the number of singular vectors used to define the deflation space to 4 vectors, problems in the training set to 12 problems and 4 eigenvectors for each matrix from the training set.

nodes	sub=1		sub=9		sub=49		sub=225	
	its	time	its	time	its	time	its	time
16^2	16	8.287	14	8.141	-	-	-	-
32^2	30	10.172	15	9.941	-	-	-	-
64^2	59	13.506	42	12.37	45	14.893	-	-
128^2	120	86.661	42	40.839	21	28.193	12	32.483

Table 4.5: Time and iteration counts (its) for Eigenvector-based deflation with varying numbers of subdomains (sub), 12 problems in the training set, 4 eigenvectors per problem in the training set, and 4 singular vectors used to define the deflation space.

Due to problems with the interface to SLEPc, we were unable to use an efficient eigenvalue solver to determined the eigenvectors needed for this test and were, unfortunately, limited to brute force iteration. This, naturally, produces very high timings for the eigenvalues needed, which skews the analysis of the resulting methods. Since this is purely a software issue (that we hope to resolve in future work), we have disregarded this time in Table 4.5, and all further results in this section; thus, the times reported are only those for the SVD computation and solves of the problems in the testing set using the balancing preconditioner. This, naturally, leads to a slightly more optimistic view of overall timing than is justified, since we do not account for one of the non-trivial costs of the algorithm.

In Table 4.5, we see an almost steady decrease in iteration count as we increase the size of the deflation space used, but not in total time. Results for the 128^2 mesh are quite interesting; with 225 subdomains, the iterations are extremely cheap, costing only about 0.37 seconds to solve each problem in the testing set. Since we focus on decreasing time per solve for the algorithms, however, we find the optimal choice of size of the deflation space for the 128^2 mesh is using 49 subdomains. Also, in the same mind set, we find the optimal sizes of the deflation spaces for the 16^2, 32^2, and 64^2 mesh is 9 subdomains, with about 0.09 seconds, 0.11 seconds, and 0.14 seconds for each solve, respectively.

We next move to the second parameter, choosing the optimal size of the training set. As in the previous method, we choose 4,8,12, and 16 problems in the training

set, while fixing 4 singular vectors used to define the deflation space and 4 eigenvectors for each problem in the training set. The iteration counts for convergence of the balancing preconditioned CG method are shown in Table 4.6.

nodes	training=4		training=8		training=12		training=16	
	its	time	its	time	its	time	its	time
16^2	14	9.03	14	8.83	14	8.29	14	7.98
32^2	15	10.88	15	10.43	15	9.94	15	9.43
64^2	42	13.62	42	13.22	42	12.37	42	12.21
128^2	22	30.69	22	30.47	22	28.19	24	29.70

Table 4.6: Time and iteration counts (its) for Eigenvector-based deflation with varying number of problems in the training set, 4 singular vectors used to define the deflation space, and 4 eigenvectors computed for each problem in the training set.

As in Solution-based deflation, the size of the training set does not greatly affect the effectiveness of the algorithms. Considering timing for the 128^2 mesh, with about 0.32 seconds per solve, the optimal size of the training set to choose for this method contains 12 vectors. This is also a reasonable choice for other problem sizes.

Now that we have fixed the size of both the deflation space and training set, we move to optimizing the next parameter, the number of eigenvectors computed for each matrix from the training set. This is done by producing t eigenvectors for each problem in the training set, and we consider values for t of 2,4,6 and 8. Iteration counts and times for these choices are shown in Table 4.7 below. The number of singular vectors used in determining the deflation space is fixed to 4 vectors. For the 128^2 mesh with 8 eigenvectors per problem in the training set and 12 training problems in total, the matrix E is of size 16384×96. Our implementation of the SVD code prevented us from calculating the singular vectors in this case.

nodes	eigen.vec=2		eigen.vec=4		eigen.vec=6		eigen.vec=8	
	its	time	its	time	its	time	its	time
16^2	13	8.29	14	8.29	13	8.37	13	8.30
32^2	15	9.91	15	9.94	14	10.10	15	10.09
64^2	42	12.48	42	12.37	42	12.78	43	13.17
128^2	24	29.46	21	28.19	22	30.31	-	-

Table 4.7: Time and iteration counts (its) for Eigenvector-based deflation with varying number of eigenvectors computed for each problem in the training set with 12 training problems and 4 singular vectors used to determine the deflation space.

In these results, since SLEPc produces all eigenvectors and singular vectors at once with each run of the method, the eigenvector time in the first part of the algorithm is the same regardless of the number of eigenvectors we request. We hope to fix this with proper tuning of our usage of SLPEc in future work. Another finding is that the SVD time increases as we increase the number of eigenvectors used, due to the increase in the size of the matrix E.

We see very little variation in performance of the balancing preconditioned CG method in this algorithm as we vary the eigenvector parameter. The variation in total time in Table 4.7 originates primarily from the SVD time. Considering the finest mesh, we see that computing 4 eigenvectors appears to be the optimal choice here.

Finally, we vary the number of singular vectors taken to determine the deflation space. We again pick the same potential numbers of vectors chosen as in the previous method. The results are shown in Table 4.8.

nodes	singular=0		singular=2		singular=4		singular=8	
	its	time	its	time	its	time	its	time
16^2	18	8.21	15	8.27	14	8.29	10	8.59
32^2	30	9.31	20	9.98	15	9.94	11	9.96
64^2	65	15.62	49	14.66	42	12.37	33	10.11
128^2	57	48.43	30	31.62	22	28.19	17	31.01

Table 4.8: Time and iteration counts (its) for Eigenvector-based deflation with varying numbers of singular vectors used to determine the deflation space with optimal numbers of problems in the training set and eigenvectors computed per problem in the training set.

The expected behaviour in Table 4.8 is that the larger the deflation space, i.e., the larger number of singular vectors chosen, the faster the CG method converges, in terms of iteration counts. However, the cost per iteration increases. We see a clear benefit to taking 4 singular vectors to define the deflation space for the 128^2 mesh, but note that more vectors improve performance on the 64^2 mesh where fewer subdomains are used.

In conclusion, we have found that the best parameters chosen for this method are to use 9 subdomains for the 16^2, 32^2, and 64^2 meshes and 49 subdomains for the 128^2 mesh. The optimal number of problems in the training set is 12 problems, with 4 eigenvectors computed for each problem in the training set, making a total of 48 eigenvectors for this method. The optimal number of singular vectors taken in the deflation space is found to be 4 singular vectors.

4.3 Conclusions and Future work:

The best performance for each method is shown in Table 4.9, although we again note that the timings recorded for eigenvector-based deflation neglect the important contribution of the time needed to compute the eigenvectors themselves.

	AMG		Solution-based		Eigenvector-based	
nodes	its	time	its	time	its	time
16^2	6	11.3	15	10.73	14	8.29
32^2	7	15	31	12.16	15	9.94
64^2	7	29.6	44	18.81	42	12.37
128^2	8	97.1	51	68.57	22	28.19

Table 4.9: Optimal total time and iteration counts for algorithm 1, 2 and 3.

In this thesis, our main goal was to construct 2 variants on the deflation methodology that efficiently solve the model problem with varying permeability fields. We compare the cost of these algorithms to that of using only an AMG preconditioned CG method. Although our algorithms are more complex in construction than AMG, we have found them to achieve our purpose. Through the use of eigenvectors, singular vectors, and deflation-based balancing preconditioners, we have found that we were able to greatly decrease the total time spent for solving the problem per permeability field from about 1 second per solve to about 0.28 seconds per solve for a mesh of size 128^2. A steady and clear decrease in time is evident in all our results in Table 4.9. In particular, the solution-based deflation algorithm is efficient but not as efficient as eigenvector-based deflation. While the number of iterations needed for the CG method to converge increases in both of our proposed algorithms in comparison to AMG, this can be considered a good trade off since the iteration counts stay low, and the cost per iteration decreases significantly. These results show insight into the power of information obtained from a small set of the permeability fields in effectively enhancing solvers for these model problems. We note that similar research has been undertaken in [11], but that work considers fixed permeability fields and a time-dependent porous media model problem.

Many possible extensions to our algorithms were not included in this work due to lack of time. For example, we were limited to generating permeability fields on meshes of size 128^2 and smaller. This limitation arises due to the way we have implemented the code, but we are aware of a more scalable algorithm to do this [28].

In this thesis, all meshes constructed on the finest level are square meshes with

equal numbers of nodes in both the x and y directions. We would like to repeat our testing on rectangular meshes where the ratio of nodes between the x and y directions is 3:1. We might see different results, or this construction may not affect the outcome.

Another direction for future work is to try implementing our two methods for GPU computing since deflation is very GPU friendly. Effective GPU processing would require dividing the method of solution of our problem into very small blocks of data with very few degrees of freedom, efficiently accessing the memory of the cluster of processors. This is denoted as "fine-scale parallelism" [3]. Parallel coding is a powerful tool that might decrease the time seen in our work in serial particularly when extending to larger problem sizes. A very important advantage we may use is the ability to parallelize the incomplete LU factorization which we have seen in this work to play a big role in the construction of our algorithms. This work has recently been shown to be possible in [9, 29]. For more information on incorporating parallelism into similar work see [1, 18, 19].

Bibliography

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, *Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects*, J. Phys.: Conf. Series, 180 (2009), p.12037.

[2] J. Bear, *Dynamics of Fluids in Porous Media*, Dover Publications Inc., 1972.

[3] N. Bell, S. Dalton, and L. Olson, *Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods*, SIAM J. Sci. Comp., 34 (2012), pp. C123–C152.

[4] J. H Bramble, *Multigrid methods*, Pitman Research Notes in Mathematics, V. 294, John Wiley and Sons, 1993.

[5] A. Brandt, *Multi-Level Adaptive Solutions to Boundary-Value Problems*, Math. Comp. 31 (1977), no. 138, p.p 333–390.

[6] A. Brandt and O.E. Livne, *Multigrid Techniques: 1984 Guide with Applications in Fluid Dynamics*, Revised edition of the 1984 original [MR0772748]. Classics in Applied Mathematics, 67. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2011. xx+218 pp.

[7] W.L. Briggs, S.F. McCormick, V.E. Henson, *A multigrid tutorial. Second edition*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.

[8] Z. Chen, G. Huan, Y. Ma, *Computational Methods for Multiphase Flows in Porous Media*, Computational Science and Engineering, 2. Society for Industrial and Applied Mathematics(SIAM), Philadelphia, PA, 2006. xxx+531 pp.

[9] E. Chow and A. Patel, *Fine-Grained Parallel Incomplete LU Factorization*, SIAM J. Sci. Comp., 37 (2015), pp. C169–C193

[10] T Cui, J Martin, Y M Marzouk, A Solonen, A Spantini, *Likelihood-informed dimension reduction for nonlinear inverse problems*, Inverse Problems 30 (2014), no. 11, 114015, 28 pp.

[11] G. B. Diaz Cortes, C. Vuik, J. D. Jansen, *On POD-based Deflation Vectors for DPCG applied to porous media problems.* , J. Comput. Appl. Math. 330 (2018), 193–213. 65M22 (76S05).

[12] J.E. Dendy, *Black box Multigrid*, J. Comput. Phys. 48 (1982), no. 3, pp. 366–386.

[13] H.C Elman and V. Forstall, *Preconditioning techniques for reduced basis methods for parameterized elliptic partial differential equations,* Society for Industrial and Applied Mathematics, 2015, Vol. 37, No. 5, pp. S177–S194.

[14] R. Eymard, R. Gallout, T. R. Herbin, *The finite volume method, Handbook of Numerical Analysis*, Vol. VII, 2000, p. 713–1020. Editors: P.G. Ciarlet and J.L. Lions.

[15] R.D. Falgout and U.M. Yang, *Hypre: a Library of High Performance Preconditioners*, Computational Science - ICCS 2002: International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part III, Springer-Verlag.

[16] G.H.Golub, C.F.Van Loan *Matrix computations, third edition*, Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 1996.

[17] D. Gottlieb and S. Orzag, *Numerical Analysis of Spectral Methods : Theory and Applications*, CBMS-NSF Regional Conference Series in Applied Mathematics, No. 26. Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1977. v+172 pp.

[18] R. Gupta and D. Lukarski and M.B. van Gijzen and C. Vuik, *Evaluation of the Deflated Preconditioned CG method to solve Bubbly and Porous Media Flow Problems on GPU and CPU*, International Journal for Numerical Methods in Fluids, 80, pp. 666–683, 2016.

[19] R. Gupta and M.B. van Gijzen and C. Vuik, *Efficient two-level preconditioned conjugate gradient method on the GPU High Performance Computing for Computational Science*, VECPAR 2012 10th International Conference, Kope, Japan, July 17-20, 2012, Revised Selected Papers Editors: M. Dayde, O. Marques, K. Nakajima Lecture Notes in Computer Science, Volume 7851 Springer, Berlin, pp. 36–49, 2013.

[20] V.E. Henson and U.M. Yang, *BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner*, Applied Numerical Mathematics, Vol. 41, pages 155-177, 2002.

[21] M. Hestenes, E.Stiefel , *Methods of conjugate gradients for solving linear systems*, J. Research Nat. Bur. Standards 49 (1952), 409–436 (1953).

[22] M. Hestenes, *Conjugate direction methods in optimization*, Applications of Mathematics, 12. Springer-Verlag, New York-Berlin, 1980. x+325 pp.

[23] A.S. Householder, *The theory of matrices in numerical analysis*, Blaisdell Publishing Co. Ginn and Co. New York-Toronto-London 1964 xi+257 pp.

[24] D. Hutton, *Fundamentals of finite element analysis*, McGraw Hill publishing Inc., New York, 2004.

[25] T.B Jönsthövel, *The deflated preconditioned conjugate gradient method applied to composite material*, Ph.D. Thesis, Delft University of Technology, The Netherlands, 2012.

[26] T.B Jönsthövel, M.B. van Gijzen, C.Vuik, C. Kasbergen, A. Scarpas, *On the use of rigid body modes in the deflated preconditioned conjugate gradient method*, SIAM J. Sci. Comput. 35 (2013), no. 1, B207–B225.

[27] R. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, Cambridge University Press, Cambridge, 2002. xx+558 pp.

[28] F. Lindgren and H. Rue, *An explicit link between Gaussian fields and Gaussian Markov random fields: the stochastic partial differential equation approach*, J. R. Stat. Soc. Ser. B Stat. Methodol. 73 (2011), no. 4, pp. 423–498.

[29] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, *Efficient Sparse Matrix-Vector Multiplication on x86-based Many-Core Processors*, in Proc. HPC Netw., Stor. Anal., SC 13, ACM, 2013, pp. 273–282.

[30] S. P. MacLachlan, *Iterative Methods in Numerical Linear Algebra* , Math 6204 Lecture notes, Memorial University.

[31] S. P. MacLachlan, *Numerical Solutions of Differential equations*, Math 6210 Lecture notes, Memorial University.

[32] S. P. MacLachlan, L. N. Olson, *Theoretical Bounds for Algebraic Multigrid Performance: Review and Analysis*, Numer. Linear Algebra Appl. 21 (2014), no. 2, p.p 194–220.

[33] S.P. MacLachlan,J.D. Moulton, and T.P. Chartier , *Robust and Adaptive Multigrid Methods: comparing structured and algebraic approaches*, Numer. Linear Algebra Appl. 19 (2012), no. 2, pp. 389–413.

[34] S. F. McCormick, (Editor) *Multigrid methods*, Chapter 1: *Introduction*, W.Briggs and S. McCormick, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA 1987, pp. xvii + 275.

[35] J.W. Ruge and K. Stüben, *Algebraic multigrid (AMG)*, in Multigrid Methods, S. F. McCormick, ed., vol. 3 of Frontiers in Applied Mathematics, SIAM, Philadelphia, PA, 1987, pp. 73–130.

[36] G. Meurant, *The Lanczos and conjugate gradient algorithms. From theory to finite precision computations*, Software, Environments, and Tools, 19. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2006. xvi+365 pp.

[37] J.N Reddy, *On the numerical solution of differential equations by the finite element method. I. An Introduction to the Finite Element Method*, (the Ritz models), Indiana J. Pure Appl. Math. 16 (1985), no. 11, 1341–1376.

[38] Y. Saad, *Iterative methods for sparse linear systems,Second edition*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2003. xviii+528 pp.

[39] Y. Saad, *Numerical methods for large eigenvalue problems. Algorithms and Architectures for Advanced Scientific Computing*, Manchester University Press, New York,1992.

[40] J.M. Tang, S.P. MacLachlan, R. Nabben, C. Vuik, *A comparison of two-level preconditioners based on multigrid and deflation*, SIAM J. Matrix Anal. Appl. 31 (2009/10), no. 4, pp. 1715–1739.

[41] S. Sickel, *A Comparison of Some Iterative Methods in Scientific Computing*, Summer Research Apprentice Program, University of Wyoming, USA, 2005.

[42] D. M. Strong, *"Iterative Methods for Solving Ax=b,"* Convergence (July 2005), url: https://www.maa.org/press/periodicals/loci/joma/iterative-methods-for-solving-iaxi-ibi.

[43] J.M. Tang, *Two-Level Preconditioned Conjugate Gradient Methods with Applications to Bubbly Flow Problems*, Ph.D. Thesis, Delft University of Technology, The Netherlands, 2008.

[44] U. Trottenberg, C. Oosterlee, A. Schuller, *Multigrid. With contributions by A. Brandt, P. Oswald and K. Stüben,* Academic Press, Inc., San Diego, CA, 2001. xvi+631 pp.

[45] A. Van der Sluis, H. A. Van der Vorst, *The rate of convergence of conjugate gradients*, Numer. Math. 48 (1986), no. 5, 543560.

[46] H.A. Van Der Vorst, *Iterative Krylov methods for large linear systems*, Cambridge Monographs on Applied and Computational Mathematics, 13. Cambridge University Press, Cambridge, 2003.

[47] R. Varga, *Matrix iterative analysis*, Prentice-Hall, Inc., Englewood Cliffs, N.J. 1962 xiii+322 pp.

[48] D.M Young, *Iterative solutions of large linear systems*, Academic Press, New York-London, 1971. xxiv+570 pp.

[49] M. S. Alnaes, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes and G. N. Wells, *The FEniCS Project Version 1.5*, Archive of Numerical Software, vol. 3, 2015.

[50] S. Abhyankar, J. Brown, E.M. Constantinescu, D. Ghosh, B.F. Smith, and H. Zhang, *PETSc/TS: A Modern Scalable ODE/DAE Solver Library*, arXiv preprint arXiv:1806.01437, 2018.

[51] V. Hernandez, J. E. Roman, and V. Vidal. *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems.* ACM Trans. Math. Software, 31(3):351-362, 2005.